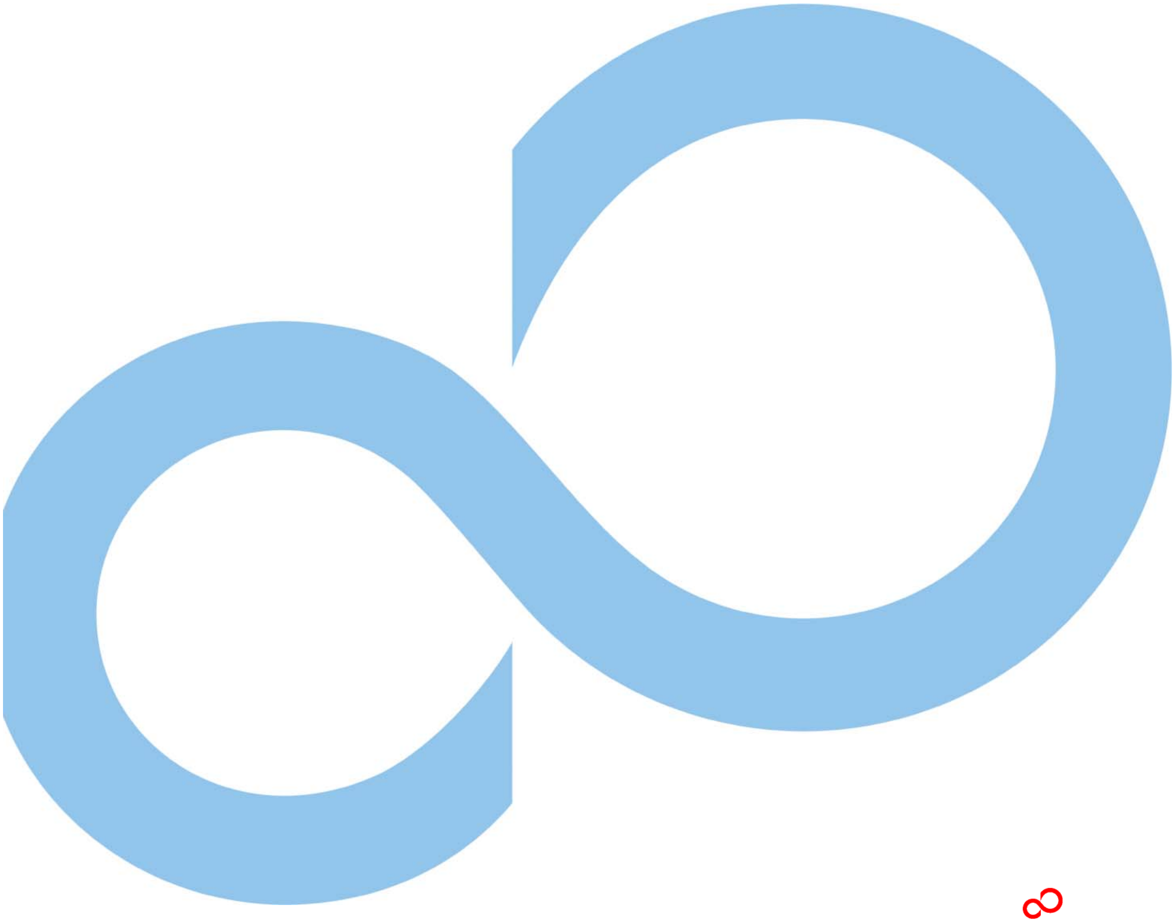
J2X0-1634-01EN

SymfoWARE(R)Server SQL Beginner's Guide



FUJITSU

Preface

■Purpose

This manual is a beginner's guide for the SymfoWARE Server Structured Query Language (SQL). The purpose of this manual is to help readers write programs for a variety of different data operations using SymfoWARE/RDB databases.

SymfoWARE is a database system supporting a client-server application configuration. In this manual, the SymfoWARE client function is called SymfoWARE client, and the SymfoWARE server function is called SymfoWARE server.

■Intended Readers

This manual is intended for users who develop application programs for processing SymfoWARE/RDB databases. Users should have the following skills and knowledge before reading this manual.

- Familiarity with SymfoWARE/RDB functions and a general understanding of databases
- Familiarity with applications that use SymfoWARE/RDB
- Ability to develop application programs using C or COBOL
- Ability to use the Solaris system or Windows N
- Ability to use Windows

■Organization of This Manual

This manual is organized as follows:

◆Chapter 1 Overview of SQL

Describes the types of SQL and explains the rules common to SQL statements in this manual.

◆Chapter 2 Retrieving Data

Explains how to use the single-row SELECT statement to retrieve data.

◆Chapter 3 Modifying Data

Explains how to use the INSERT statement, UPDATE statement (search), and DELETE statement (search) to modify data.

◆Chapter 4 Using a Cursor to Manipulate Data

Explains how to manipulate data using a cursor.

◆Chapter 5 Joining Multiple Tables and Manipulating Data

Explains how to join and manipulate data in multiple tables.

◆Chapter 6 Methods of Manipulating Data

Describes the methods used to manipulate data.

◆Chapter 7 Executing Dynamic SQL Statements while Application Programs Are Running

Explains how to execute dynamic SQL statements while application programs are running.

◆Appendix A SQL Data Types and Equivalent Host Variable Data Types

Describes SQL data types and their equivalent host variable data types.

◆Appendix B Handling RDB Messages

Explains how to reference user responses to messages displayed by SymfoWARE/RDB.

◆Appendix C SQLSTATE Values

Explains the SQLSTATE values reported to application programs as processing results.

◆Glossary

The glossary contains definitions of the terms used in this manual.

■How to Use This Manual

This manual is intended as reference material for users developing application programs using SymfoWARE/RDB.

First-time users of SymfoWARE/RDB should read the RDB User's Guide: Database Definition and the RDB User's Guide: Application Program Development before reading this manual. These manuals provide readers with an overview of SymfoWARE/RDB databases and the functions of SQL statements. These manuals also provide an overview on specifying SQL statements and developing application programs. For an explanation on how to develop application programs using SymfoWARE Programmer's Kit, refer to the Esql Programmer's Guide.

This manual uses concrete examples to explain many kinds of data manipulation using SQL statements. Readers do not need to master the entire contents of this manual all at once. Instead, read through the manual to gain an overview of the kinds of processing that can be performed using SQL statements. Then, reread the parts of this manual related to writing application programs. In explanations of the syntax of SQL statements, this manual emphasizes ease of understanding rather than giving precise details about the syntax. For details about the syntax of SQL statements, refer to the SQL Reference Guide.

Examples of specifying SQL statements given in this manual are written in C unless otherwise specified. Keywords in the explanations appear in Gothic type. In addition, standard terms are added to the descriptions of SQL statements. Refer to these terms when referring to the reference manual.

■Related manuals

The table below lists related manuals and their informal names used in this manual.Informal name in this manual

SymfoWARE Server

Notation in this manual	Manual title for each OS	
	Solaris	Windows NT
General Description	FUJITSU SymfoWARE Server General Description	
Start Guide: Client	FUJITSU SymfoWARE Server Start Guide: Client	
Start Guide: Server	–	FUJITSU SymfoWARE Server Start Guide: Server (EE/SE)
RDB Operations Guide	SymfoWARE Server RDB Operations Guide	SymfoWARE Server Operations Guide (EE/SE)
RDB User's Guide: Database Definition	FUJITSU SymfoWARE Server RDB User's Guide: Database Definition	
RDB User's Guide: Application Program Development	FUJITSU SymfoWARE Server RDB User's Guide: Application Program Development	
SQL Beginner's Guide (This manual)	FUJITSU SymfoWARE Server SQL Beginner's Guide	
SQL Reference Guide	FUJITSU SymfoWARE Server SQL Reference Guide	
SQLTOOL User's Guide	FUJITSU SymfoWARE Server SQLTOOL User's Guide	
RDA-SV Operations Guide	SymfoWARE Server RDA-SV Operations Guide	
Cluster Installation/ Administration Guide	SymfoWARE Cluster Installation/Administration Guide	–

· No corresponding manual

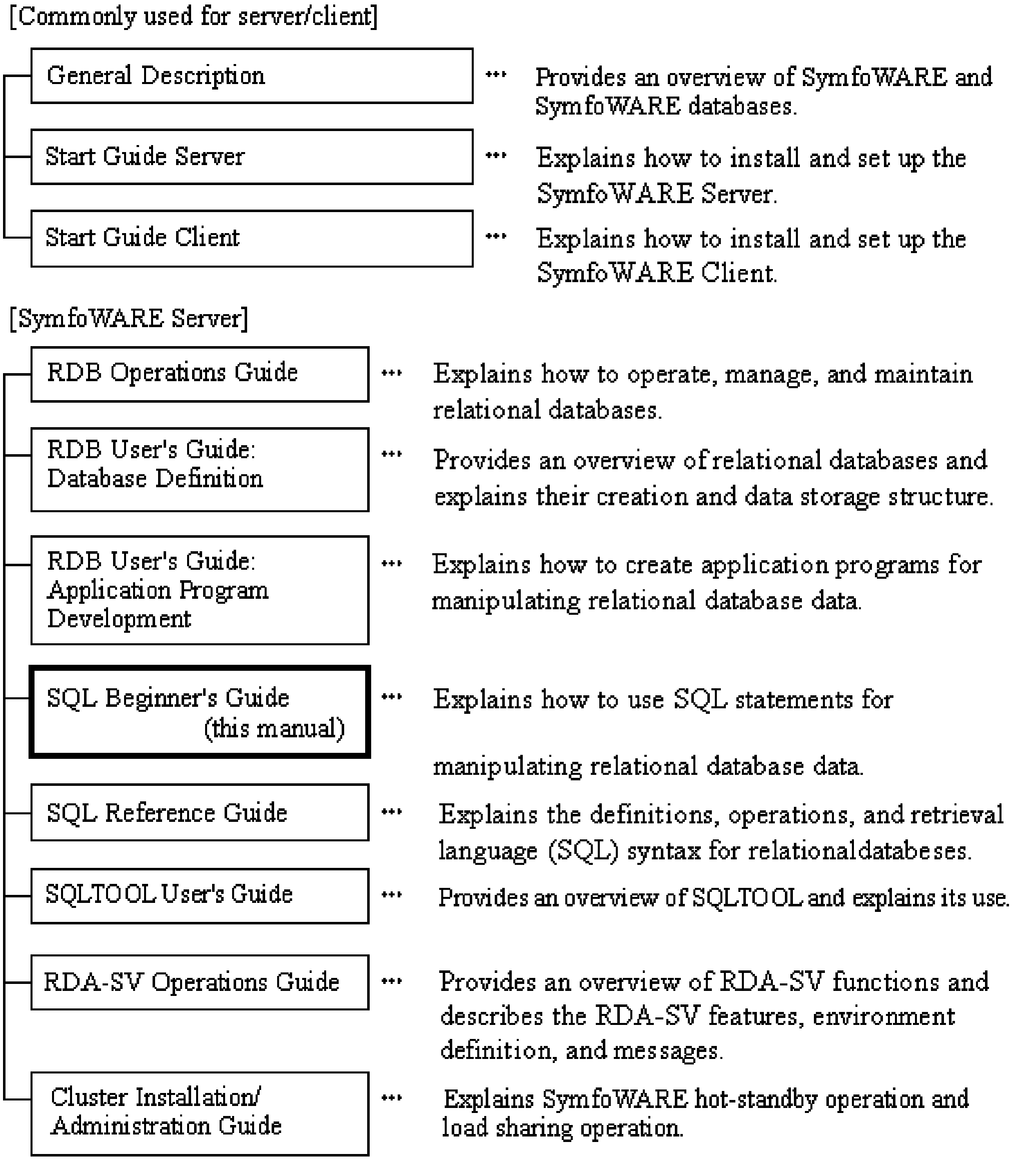
SymfoWARE Programmer's Kit

Notation in this manual	Manual title
RDB User's Guide: Application Development	FUJITSU SymfoWARE Server RDB User's Guide: Application Program Development
SQL Beginner's Guide (Online manual) (This manual)	FUJITSU SymfoWARE Server SQL Beginner's Guide
SQL Reference Guide (Online manual)	FUJITSU SymfoWARE Server SQL Reference Guide
SymfoWARE/RDB Messages (Online manual)	SymfoWARE/RDB Message Reference
SymfoWARE/RDB Commands (Online manual)	SymfoWARE Programmer's Kit RDB Commands Reference
JDBC Drivers (Online manual)	SymfoWARE Client JDBC Drivers Online Manual

■Position of this manual

The manual system and the position of this manual are as follows:

SymfoWARE Server



[Commonly used for server/client]

Manual	Description
General Description	··· Provides an overview of SymfoWARE and SymfoWARE databases.
Start Guide Server	··· Explains how to install and set up the SymfoWARE Server.
Start Guide Client	··· Explains how to install and set up the SymfoWARE Client.

[SymfoWARE Server]

Manual	Description
RDB Operations Guide	··· Explains how to operate, manage, and maintain relational databases.
RDB User's Guide: Database Definition	··· Provides an overview of relational databases and explains their creation and data storage structure.
RDB User's Guide: Application Program Development	··· Explains how to create application programs for manipulating relational database data.
SQL Beginner's Guide (this manual)	··· Explains how to use SQL statements for manipulating relational database data.
SQL Reference Guide	··· Explains the definitions, operations, and retrieval language (SQL) syntax for relationaldatabeses.
SQLTOOL User's Guide	··· Provides an overview of SQLTOOL and explains its use.
RDA-SV Operations Guide	··· Provides an overview of RDA-SV functions and describes the RDA-SV features, environment definition, and messages.
Cluster Installation/ Administration Guide	··· Explains SymfoWARE hot-standby operation and load sharing operation.

In addition to the preceding manuals, SymfoWARE/RDB provides an online manual.

Displaying Command References

UNIX

The man command is used to display the syntax of RDB commands.

For details on the man command, refer to AnswerBook2 of the Reference Manual Collection.

The copyright of the online manual is the property of UNIX System Laboratories, Inc. and Fujitsu Limited. Follow the items in the written contract to use this product properly

Windows NT/2000/XP

See the online help of Windows NT/2000/XP

Action in response to displayed messages

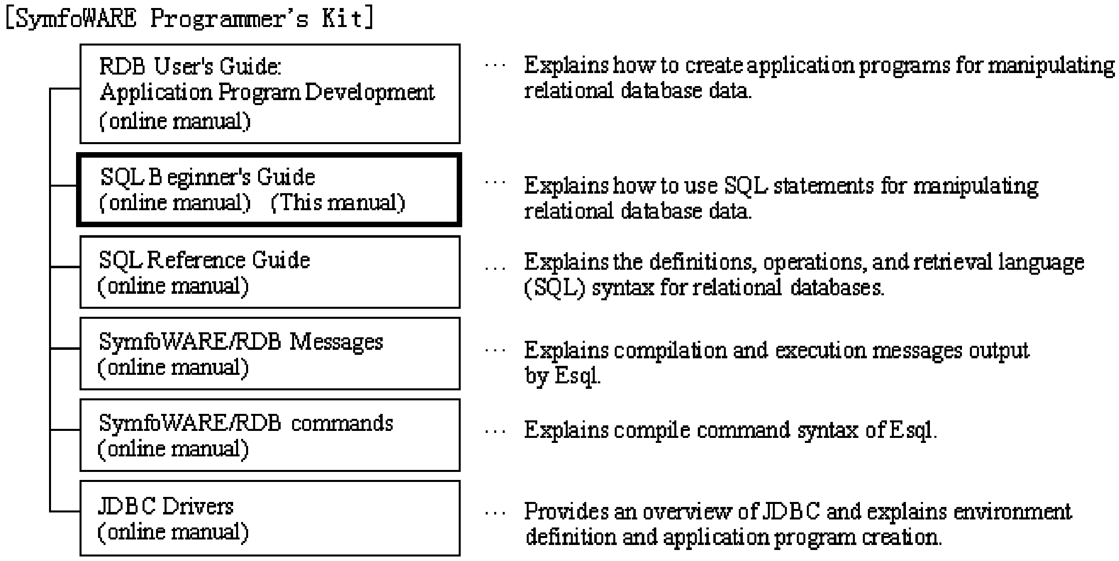
UNIX

The rdbprtmsg command (RDB command) gives the meaning and user response for each displayed message.

Windows NT/2000/XP

See the online help of Windows NT/2000/XP.

◆SymfoWARE Programmer's Kit



[SymfoWARE Programmer's Kit]

RDB User's Guide: Application Program Development (online manual)	… Explains how to create application programs for manipulating relational database data.
SQL Beginner's Guide (online manual) (This manual)	… Explains how to use SQL statements for manipulating relational database data.
SQL Reference Guide (online manual)	… Explains the definitions, operations, and retrieval language (SQL) syntax for relational databases.
SymfoWARE/RDB Messages (online manual)	… Explains compilation and execution messages output by Esql.
SymfoWARE/RDB commands (online manual)	… Explains compile command syntax of Esql.
JDBC Drivers (online manual)	… Provides an overview of JDBC and explains environment definition and application program creation.

■Related manuals

The related manuals are as follows:

- Reference Manuals Collection of AnswerBook 2
- Fujitsu COBOL User's Guide for Windows
- COBOL85 User's Guide
- Fujitsu COBOL Language Reference

■Precautions

◆Applicable products

UNIX

- SymfoWARE Server Enterprise Edition 5.0 or later
- SymfoWARE Server Hot-Standby Options 5.0 or later

Windows NT, Windows 2000, or Windows XP

- SymfoWARE Server Enterprise Edition V5.0 L10 or later

SymfoWARE Programmer's Kit

- SymfoWARE Programmer's Kit V5.0 L10

◆Operating systems supporting SymfoWARE/RDB

- Solaris
- Microsoft(R) Windows NT(R) Server, Enterprise Edition
- Microsoft(R) Windows NT(R) Server network operating system
- Microsoft(R) Windows NT(R) Workstation operating system
- Microsoft(R) Windows NT(R) Server, Terminal Server Edition
- Microsoft(R) Windows(R) 2000 Professional operating system
- Microsoft(R) Windows(R) 2000 Server operating system
- Microsoft(R) Windows(R) 2000 Advanced Server operating system

- Microsoft(R) Windows XP Professional

◆Operating systems supporting SymfoWARE Programmer's Kit

- Microsoft(R) Windows(R) 95 operating system
- Microsoft(R) Windows(R) 98 operating system
- Microsoft(R) Windows(R) 98 Second Edition
- Microsoft(R) Windows(R) Millennium Edition
- Microsoft(R) Windows XP Professional
- Microsoft(R) Windows XP Home Edition
- Microsoft(R) Windows NT(R) Server, Enterprise Edition
- Microsoft(R) Windows NT(R) Server network operating system
- Microsoft(R) Windows NT(R) Workstation operating system
- Microsoft(R) Windows NT(R) Server, Terminal Server Edition
- Microsoft(R) Windows(R) 2000 Professional operating system
- Microsoft(R) Windows(R) 2000 Server operating system
- Microsoft(R) Windows(R) 2000 Advanced Server operating system

◆UNIX release version

This system conforms to UNIX System V Rel 4.2MP.

◆Print examples in this manual

Althouth printed examples of SymfoWARE/RDB in this manual are not complete, readers can figure out the image of printed example.

◆Explanatory models

Retail store stock management databases are the main models for the databases used in the examples in this manual. The data in these databases is fictitious.

■Abbreviations

The table below lists the abbreviations used in this manual:

Abbreviation	Full name
Windows 95/98/Me	Microsoft(R) Windows(R) 95 operating system Microsoft(R) Windows(R) 98 operating system Microsoft(R) Windows(R) 98 Second Edition Microsoft(R) Windows(R) Millennium Edition Microsoft(R) Windows(R) XP Professional Microsoft(R) Windows(R) XP Home Edition
Windows	Microsoft(R) Windows(R) 95 operating system Microsoft(R) Windows(R) 98 operating system Microsoft(R) Windows(R) 98 Second Edition Microsoft(R) Windows(R) Millennium Edition Microsoft(R) Windows(R) XP Professional Microsoft(R) Windows(R) XP Home Edition Microsoft(R) Windows NT(R) Server network operating system Microsoft(R) Windows NT(R) Workstation operating system Microsoft(R) Windows NT(R) Server, Enterprise Edition Microsoft(R) Windows NT(R) Server, Terminal Server Edition Windows(R) 2000 Professional operating system Windows(R) 2000 Server operating system Microsoft(R) Windows(R) 2000 Advanced Server operating system
Windows NT	Microsoft(R) Windows NT(R) Server network operating system Microsoft(R) Windows NT(R) Workstation operating system Microsoft(R) Windows NT(R) Server, Enterprise Edition Microsoft(R) Windows NT(R) Server, Terminal Server Edition
Windows 2000	Microsoft(R) Windows(R) 2000 Professional operating system Microsoft(R) Windows(R) 2000 Server operating system Microsoft(R) Windows(R) 2000 Advanced Server operating system
Windows XP	Microsoft(R) Windows(R) XP Professional Microsoft(R) Windows(R) XP Home Edition
Windows NT/2000/XP	Microsoft(R) Windows NT(R) Server network operating system Microsoft(R) Windows NT(R) Workstation operating system Microsoft(R) Windows NT(R) Server, Enterprise Edition Microsoft(R) Windows NT(R) Server, Terminal Server Edition Microsoft(R) Windows(R) 2000 Professional operating system Microsoft(R) Windows(R) 2000 Server operating system Microsoft(R) Windows(R) 2000 Advanced Server operating system Microsoft(R) Windows(R) XP Professional Microsoft(R) Windows(R) XP Home Edition
MS-DOS	Microsoft(R) MS-DOS(R) 5.0/V or later Microsoft(R) MS-DOS(R) 6.2/V or later
SymfoWARE Server	SymfoWARE Server Enterprise Edition

July 2002

Microsoft, MS, MS-DOS, Windows, and Windows NT are trademarks or registered trademarks of Microsoft Corporation in the United States and other countries.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

Solaris is a trademark of Sun Microsystems, Inc in the United States.

Lotus is a registered trademark of Lotus Development Corporation.

SymfoWARE is a registered trademark of Fujitsu Limited.

Other company and product names in this manual are trademarks or registered trademarks of their respective companies.

The (R) and TM symbols have been omitted in this manual.

All Rights Reserved, Copyright (C) FUJITSU LIMITED 2002

Chapter 1 Overview of SQL

This chapter describes the types of SQL and the rules common to SQL statements in this manual. This chapter contains the following sections:

1.1 Types of SQL

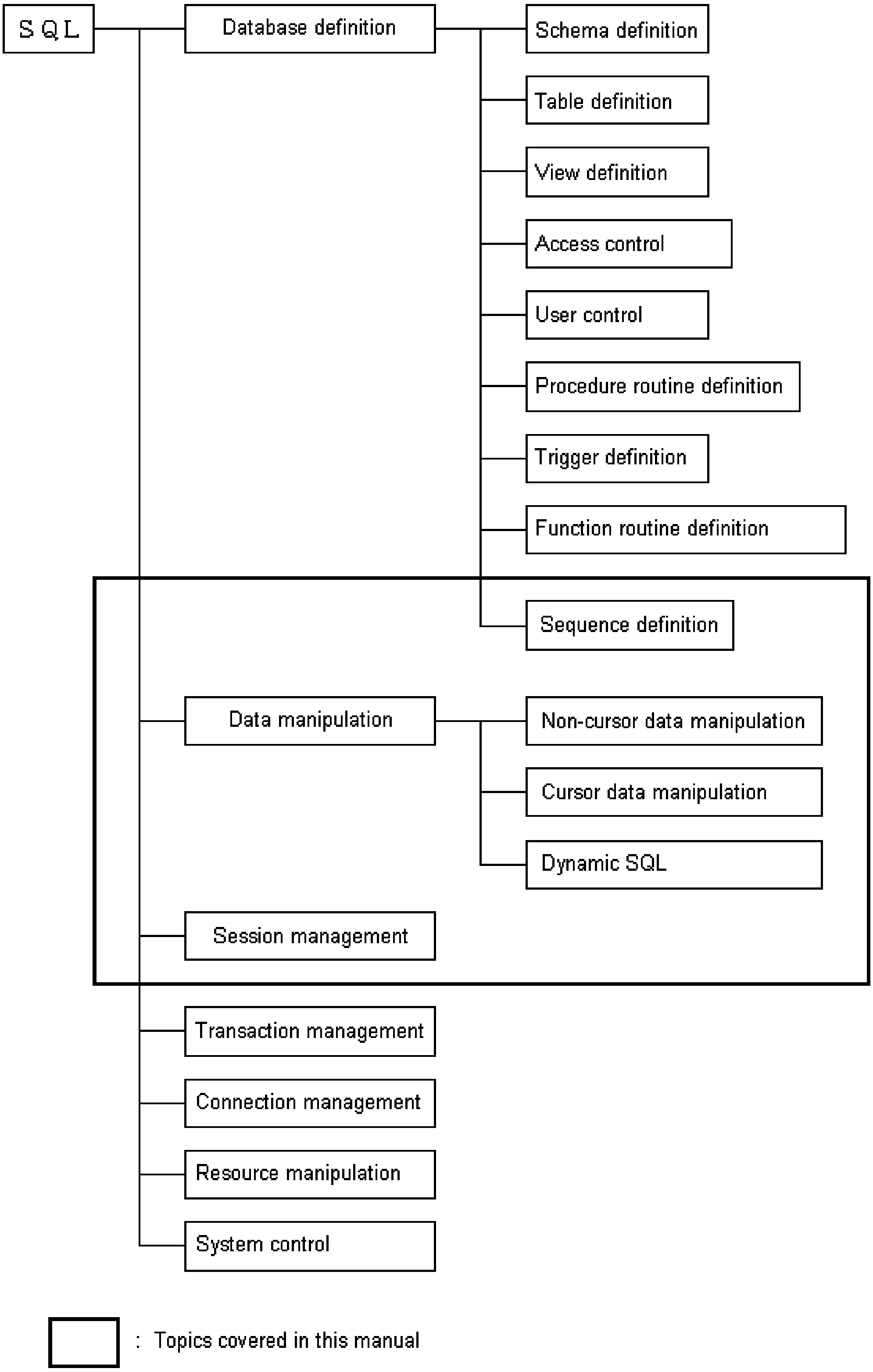
1.2 Common SQL Statement Rules

1.1 Types of SQL

SymfoWARE/RDB uses the international standard SQL for data manipulation, which is linked to languages such as C language and COBOL. SQL is a database language used for defining and accessing data bases.

The following Figure: Types of SQL shows the types of SQL for SymfoWARE/RDB:

[Figure: Types of SQL]


SQL
Database definition
Schema definition
Table definition
View definition
Access control
User control
Procedure routine definition
Trigger definition
Function routine definition
Sequence definition
Data manipulation
Non-cursor data manipulation
Cursor data manipulation
Dynamic SQL
Session management
Transaction management
Connection management
Resource manipulation
System control
: Topics covered in this manual

This manual explains how to use data manipulation SQL statements to write application programs for manipulating data and how to use session management SQL statements, required to manipulate data.

1.2 Common SQL Statement Rules

This section explains the rules common to SQL statements in this manual.

1.2.1 Names specified in SQL statements

The following names can be specified in SQL statements:

a. Schema names
b. Table names
c. Column names
d. Routine names
e. Parameter names
f. Trigger names
g. Host identifiers names
h. SQL variable names
i. Descriptor names
j. Connection names
k. SQL server names
l. Cursor names
m. Correlation names
n. SQL statement identifiers
o. Sequence names

The names indicated by a to c above are defined in the database. These names are determined when schemas and tables are designed. The names indicated by d are routine names defined in the procedure routine definition or function routine definition. The names indicated by e are the parameter names defined in the procedure routine definition. Specify these names as is in SQL statements. The names indicated by f are the trigger names defined in the trigger definition. For information about database definitions, procedure routine definitions, and trigger definitions, see the "RDB User's Guide: Database Definition."

The names indicated by g are host variable names. Host variables are named in the embedded SQL declaration clause and used in SQL statements. Host variable names are defined in C and COBOL85. These names must meet the following conditions:

- Names must not begin with SQL.

For information on the data type of host variables, see Appendix A "SQL Data Types and Equivalent Host Variable Data Types".

The names in h) are defined with compound statements consisting of procedure routine definitions.

The names in i) to n) are assigned in SQL statements. The data type for the names in i), j), and k) must be character-string type. The rules for assigning the names in l), m), and n) are as follows:

- Names must start with an alphanumeric character.
- The second and subsequent characters must be alphanumeric or numeric or an underline.
- Lowercase characters are converted to uppercase characters.
- Names can be up to 18 characters long.

1.2.2 Data base used in examples

In this section, an inventory management data base for a retail store is used as an example of a data base. The same data base is used throughout this manual in explanations of data manipulations. This data base contains the following three tables used in running inventory management applications for the retail store.

STOCK table:
This table contains information about products handled and stock quantities.
ORDER table:
This table contains information about products, the quantities of products ordered, and the prices of products by supplier.
COMPANY table:
This table contains information about names, telephone numbers, and addresses of customer companies.

These tables are contained in a schema called STOCKS.

■STOCK table

Figure: Inventory management data base a) shows the contents of the STOCK table. The STOCK table contains the following four columns:

ITMNO:
This column lists the code numbers assigned to the products.
PRODUCT:
This column lists the product names.
STOCKQTY:
This column lists the quantities of stock for the products.
WHCODE:
This column lists the numbers of the warehouses in which the products are stored.

■ORDER table

Figure: Inventory management data base b) shows the contents of the ORDER table. The ORDER table contains the following four columns:

CUSTOMER:
This column lists the customers' company numbers.
PRODNO:
This column lists the code numbers assigned to the products.
PRICE:
This column lists the prices of the products.
ORDERQTY:
This column lists the quantities of products ordered.

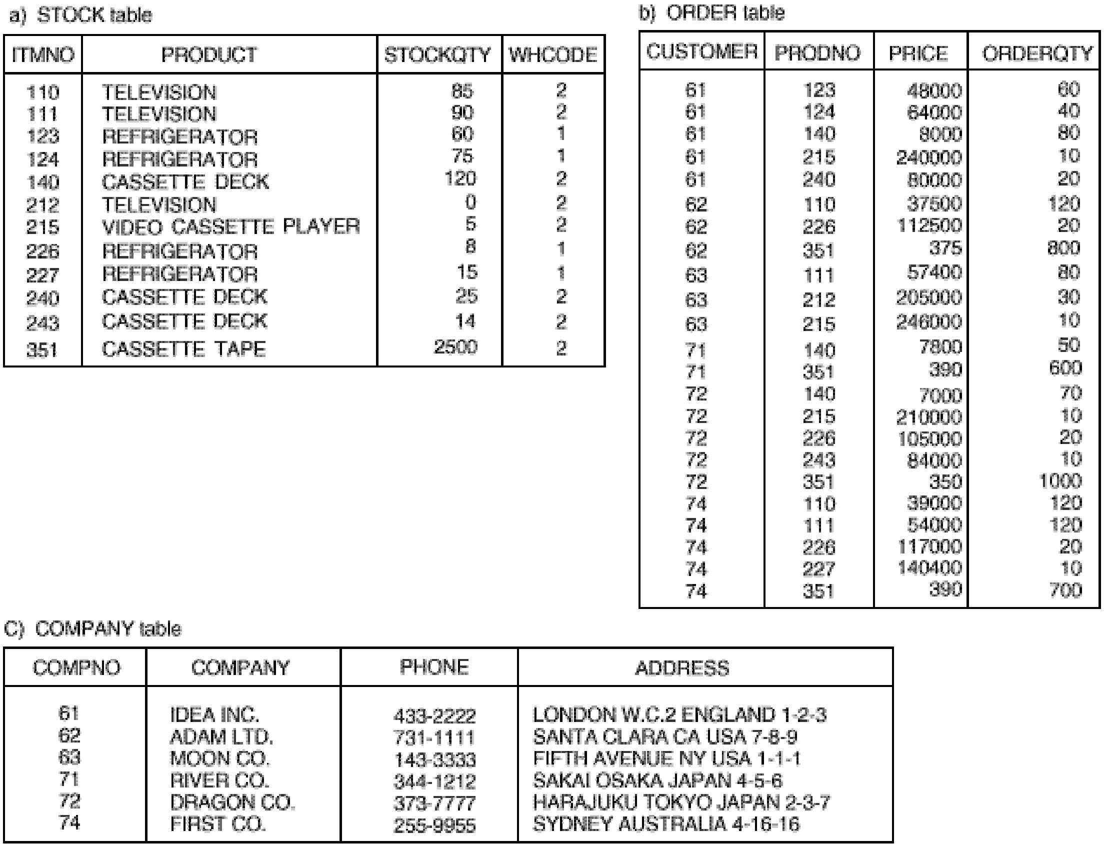
■COMPANY table

Figure: Inventory management data base c) shows the contents of the COMPANY table. The COMPANY table contains the following four columns:

COMPNO:
This column lists the code numbers assigned to the companies.
COMPANY:
This column lists the names of the companies.
PHONE:
This column lists the telephone numbers of the companies.
ADDRESS:
This column lists the addresses of the companies.

In this manual, unless otherwise stated, when the preceding table or column names are used in examples of SQL statements, the tables and columns in Figure: Inventory management data base are specified. Refer to Figure: Inventory management data base when necessary. Fictitious data is used in the tables in Figure: Inventory management data base.

[Figure: Inventory management data base]



a) STOCK table

ITMNO	PRODUCT	STOCKQTY	WHCODE
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
140	CASSETTE DECK	120	2
212	TELEVISION	0	2
215	VIDEO CASSETTE PLAYER	5	2
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETTE TAPE	2500	2

b) ORDER table

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	123	48000	60
61	124	64000	40
61	140	8000	80
61	215	240000	10
61	240	80000	20
62	110	37500	120
62	226	112500	20
62	351	375	800
63	111	57400	80
63	212	205000	30
63	215	246000	10
71	140	7800	50
71	351	390	600
72	140	7000	70
72	215	210000	10
72	226	105000	20
72	243	84000	10
72	351	350	1000
74	110	39000	120
74	111	54000	120
74	226	117000	20
74	227	140400	10
74	351	390	700

C) COMPANY table

COMPNO	COMPANY	PHONE	ADDRESS
61	IDEA INC.	433-2222	LONDON W.C.2 ENGLAND 1-2-3
62	ADAM LTD.	731-1111	SANTA CLARA CA USA 7-8-9
63	MOON CO.	143-3333	FIFTH AVENUE NY USA 1-1-1
71	RIVER CO.	344-1212	SAKAI OSAKA JAPAN 4-5-6
72	DRAGON CO.	373-7777	HARAJUKU TOKYO JAPAN 2-3-7
74	FIRST CO.	255-9955	SYDNEY AUSTRALIA 4-16-16

■Attributes of table columns in inventory management data base

Table: Attributes of table columns in the inventory management data base lists the attributes of table columns in the inventory management data base.

[Table: Attributes of table columns in the inventory management data base]

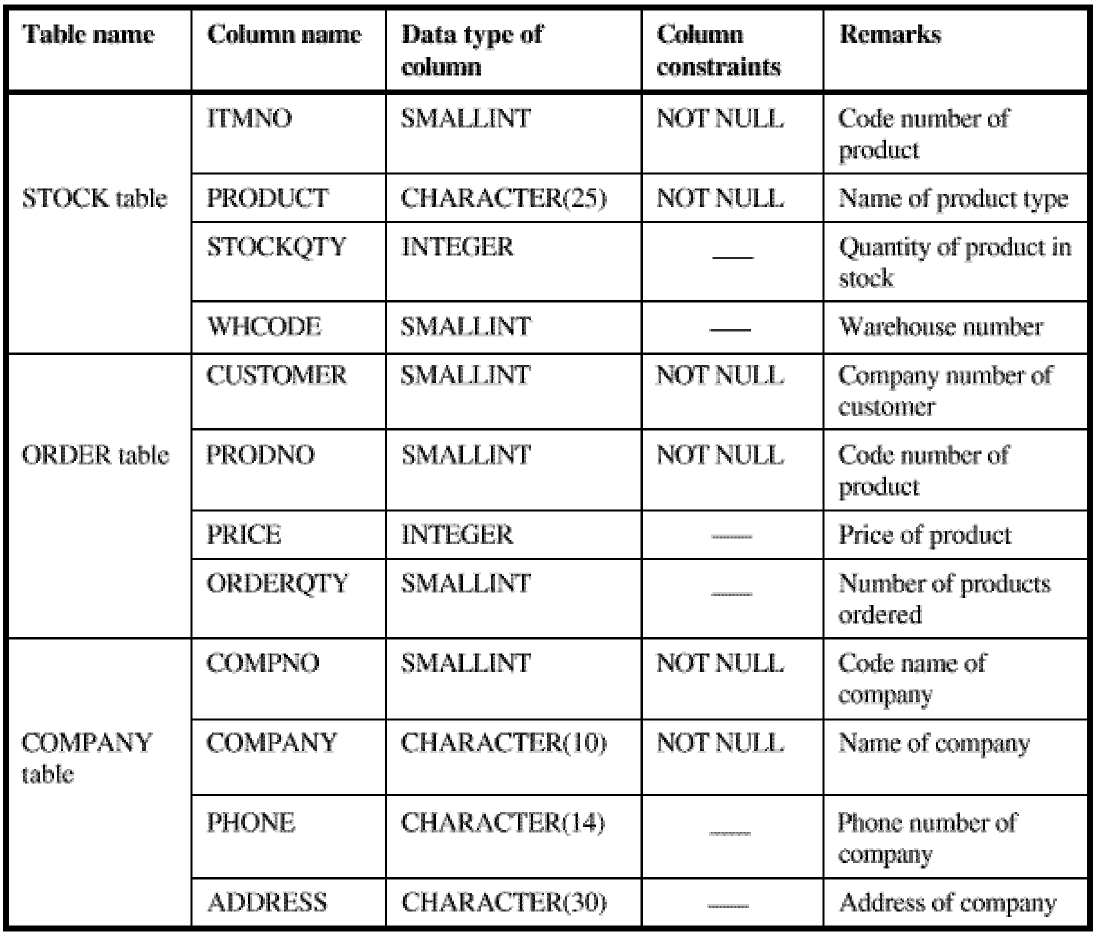


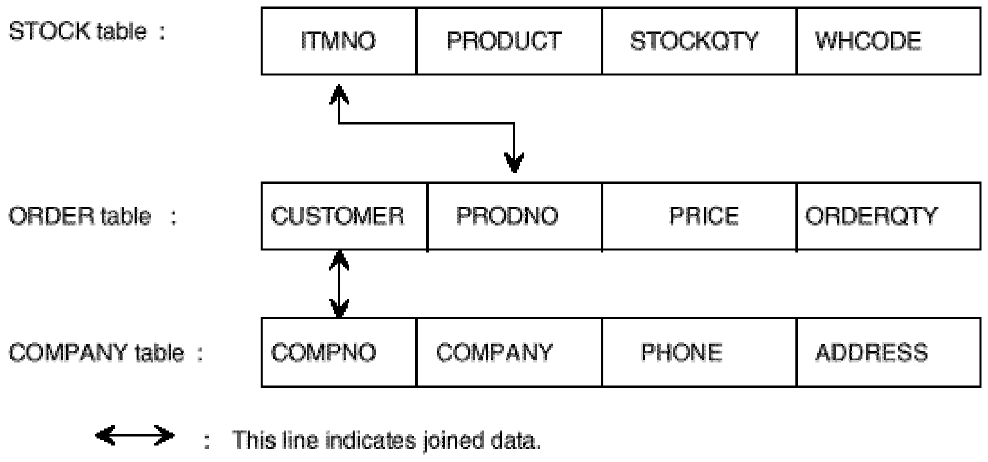
Table name	Column name	Data type of column	Column constraints	Remarks
STOCK table	ITMNO	SMALLINT	NOT NULL	Code number of product
	PRODUCT	CHARACTER(25)	NOT NULL	Name of product type
	STOCKQTY	INTEGER	—	Quantity of product in stock
	WHCODE	SMALLINT	—	Warehouse number
ORDER table	CUSTOMER	SMALLINT	NOT NULL	Company number of customer
	PRODNO	SMALLINT	NOT NULL	Code number of product
	PRICE	INTEGER	—	Price of product
	ORDERQTY	SMALLINT	—	Number of products ordered
COMPANY table	COMPNO	SMALLINT	NOT NULL	Code name of company
	COMPANY	CHARACTER(10)	NOT NULL	Name of company
	PHONE	CHARACTER(14)	—	Phone number of company
	ADDRESS	CHARACTER(30)	—	Address of company

—: NULL is allowed.

■Relationship between STOCK table, ORDER table, and COMPANY table

Figure: Relationship between STOCK table, ORDER table, and COMPANY table shows the relationship between the three tables. The STOCK table and the ORDER table are joined through ITMNO and PRODNO. The ORDER table and the COMPANY table are joined through CUSTOMER and COMPNO. For example, the product with PRODNO 123 in the STOCK table is listed as REFRIGERATOR under PRODUCT. STOCKQTY of the product is 60 and the product is in warehouse number 1. The row for PRODNO 123 in the ORDER table provides information on PRICE and ORDERQTY for this product, which is 48,000 and 60, respectively. Finally, COMPNO of CUSTOMER is 61. So, from the row for COMPNO 61 in the COMPANY table, CUSTOMER for the product and COMPANY, PHONE, and ADDRESS of the company can be ascertained.

[Figure: Relationship between STOCK table, ORDER table, and COMPANY table]


STOCK table :
ITMNO
PRODUCT
STOCKQTY
WHCODE
ORDER table :
CUSTOMER
PRODNO
PRICE
ORDERQTY
COMPANY table :
COMPNO
COMPANY
PHONE
ADDRESS
: This line indicates joined data.

Chapter 2 Retrieving Data

This chapter describes the data manipulation that can be performed using the single row SELECT statement.

This chapter contains the following sections:

2.1 Fetching Column Values without Modifications

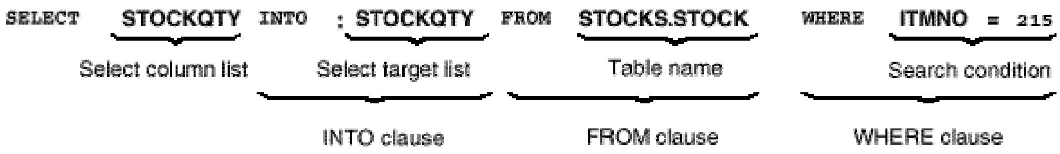
2.2 Fetching Columns that are Undefined

2.1 Fetching Column Values without Modifications

Use the single row SELECT statement to fetch one row of data from a table. The following is an example of the single row SELECT statement:

Example:

In this example, STOCKQTY of the row for ITMNO 215 is obtained from the STOCK table.



```
SELECT STOCKQTY INTO :STOCKQTY FROM STOCKS.STOCK WHERE ITMNO = 215
```

In this example, host variable STOCKQTY is specified for the select target list in the single row SELECT statement. Host variables are used to pass data between the application program and the database. That is, a host variable is specified in an SQL statement to fetch data to be accessed by the application program from the database and to store data specified by the application program in the database. An embedded SQL declaration clause is used to declare a host variable. When a host variable is specified in an SQL statement, prefix the host variable with a colon. However, for SQL statements that are not in C and for COBOL statements, specify host variables the same way as for general C and COBOL variables without a colon prefixed. For C, national-language characters cannot be used for host variable names.

This section explains the following operations that can be performed by the single row SELECT statement:

- Fetching values from multiple columns
- Fetching data that contains a null value
- Fetching data from multiple tables
- Deleting duplicate rows and fetching data

2.1.1 Fetching values from multiple columns

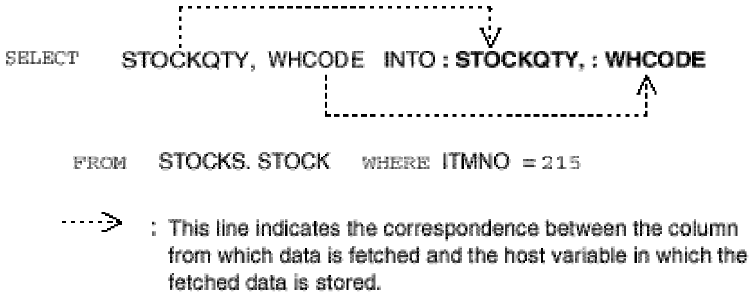
In the first example in 2.1 "Fetching Column Values without Modifications", only data from column STOCKQTY was fetched. Suppose that data from two columns, STOCKQTY and WHCODE, is to be fetched. Obviously, the single row SELECT statement could be executed twice. However, executing the statement more than once increases the number of columns from which data is to be fetched and processing efficiency deteriorates. Instead, the single row SELECT statement can be executed once to fetch values from multiple columns.

When fetching data from multiple columns, separate the column names in the select column list by commas. Specify an equal number of host variables for storing the fetched data. Data from fetched columns is stored in host variables in the same sequence from the left. Making the host variable names the same as the column names makes the application program more understandable. However, note that values are not stored in the host variables with the same names.

In the following example multiple columns are specified in a single row SELECT statement.

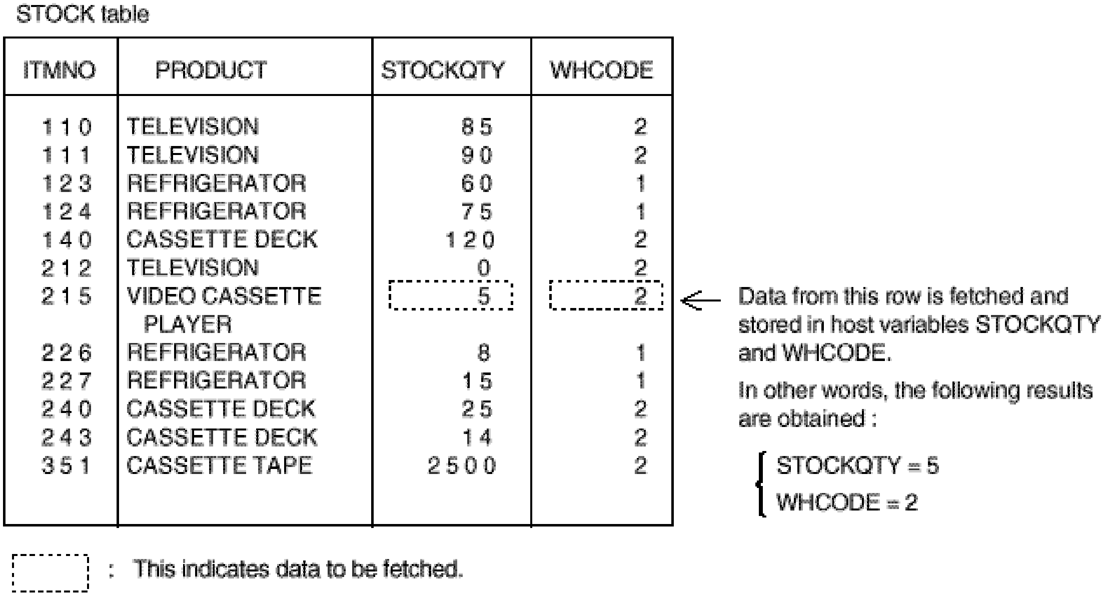
Example 1:

In this example, values for STOCKQTY and WHCODE for the product with ITMNO "215" are obtained from the STOCK table. The values obtained for STOCKQTY and WHCODE are stored in host variables STOCKQTY and WHCODE, respectively.


SELECT STOCKQTY, WHCODE INTO : STOCKQTY, : WHCODE
FROM STOCKS. STOCK WHERE ITMNO = 215

----> : This line indicates the correspondence between the column from which data is fetched and the host variable in which the fetched data is stored.

[Figure: Example of specifying multiple columns in a single row SELECT statement]



STOCK table

ITMNO	PRODUCT	STOCKQTY	WHCODE
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
140	CASSETTE DECK	120	2
212	TELEVISION	0	2
215	VIDEO CASSETTE PLAYER	5	2
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETTE TAPE	2500	2

← Data from this row is fetched and stored in host variables STOCKQTY and WHCODE.

In other words, the following results are obtained :

STOCKQTY = 5
WHCODE = 2

: This indicates data to be fetched.

If all columns in a table are to be specified in the order in which they were defined, an asterisk can be used to specify the columns instead of column names. In the following example an asterisk is used to specify columns:

Example 2:

In this example, an asterisk is used to fetch data from all columns in the STOCK table.

```
SELECT * INTO  :ITMNO,  :PRODUCT,  :STOCKQTY,  :WHCODE


          FROM  STOCKS. STOCK  WHERE ITMNO = 215
```

In Example 2, the specification of the columns from which data is to be fetched is equivalent to the four columns ITMNO, PRODUCT, STOCKQTY, and WHCODE being specified separately. Therefore, four host variables must be specified for storing data fetched from columns.

If an asterisk is specified instead of column names, columns may be added or the sequence in which columns are defined may be modified due to modifications to the design of the table. If this is the case, the application program must be modified. To prevent modifications needing to be made to the application program, specify the columns to be accessed and separate them by commas.

2.1.2 Fetching data containing null values

Fetching data containing null values was not considered in the first two examples. This section explains how to fetch data containing null values.

The variable used to indicate whether a data value is a null value is called the indicator variable. The indicator variable is paired with a host variable and specified in an SQL statement. When fetching data to be accessed by the application program from the database, specify in the indicator variable whether the data to be fetched contains null values. When storing data specified by the application program in a database, also specify in the indicator variable whether the data to be stored contains null values. Use an embedded SQL declaration clause to declare the indicator variable. The declaration format is the same as that for a host variable. When an indicator variable is specified in an SQL statement, prefix the indicator variable with a colon. However, for indicator variables for C and COBOL statements, specify indicator variables the same way as for general C and COBOL variables without a colon prefixed. For C, national-language characters cannot be used for indicator variable names. The data type for indicator variables must be a data type equivalent to SMALLINT. See Appendix A "SQL Data Types and Equivalent Host Variable Data Types", for the correspondence between data types and SQL types for each language.

The following is an example of specifying indicator variables:

Example:

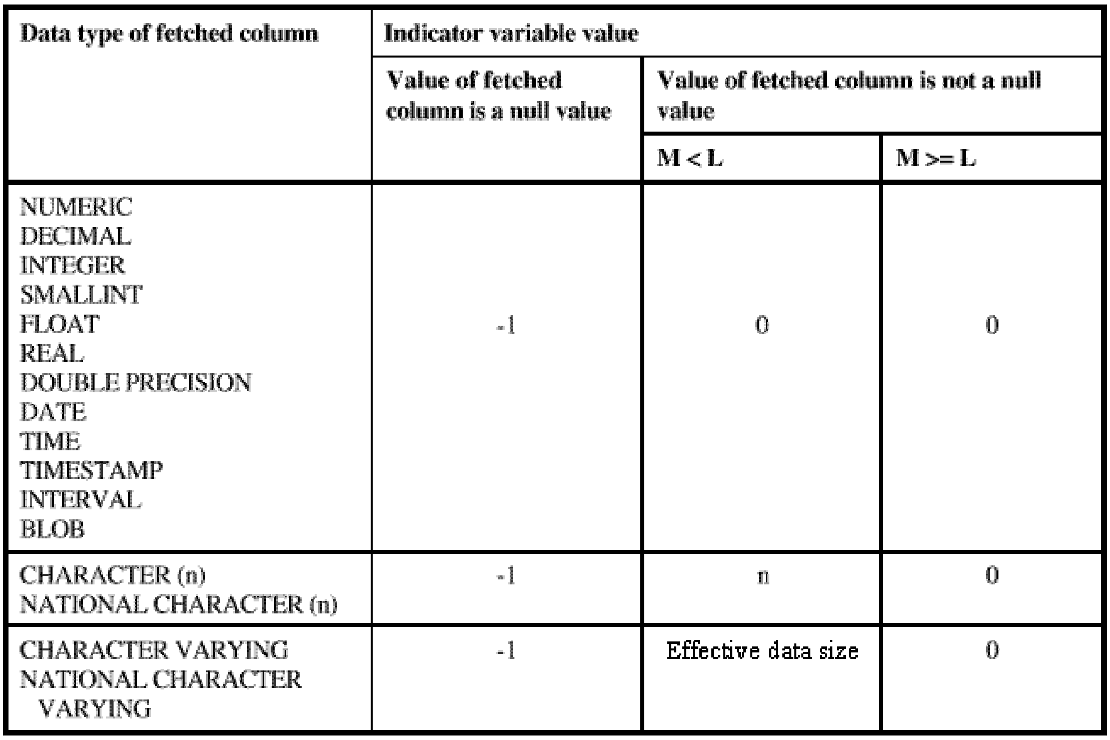
> In this example, the values for STOCKQTY and WHCODE for the product with ITMNO "215" are obtained from the STOCK table.

```
SELECT STOCKQTY, WHCODE INTO :STOCKQTY INDICATOR :STOCKQTY_INDICATOR,

                          :WHCODE INDICATOR :WHCODE_INDICATOR

      FROM STOCKS. STOCK WHERE ITMNO = 215
```

In this example, the indicator variables in the single row SELECT statement are STOCKQTY_INDICATOR and WHCODE_INDICATOR. These indicators correspond to STOCKQTY and WHCODE, respectively. Indicator variables need only be specified as required. For example, an indicator variable can be specified for WHCODE without one being specified for STOCKQTY.

Table: Column values and values stored in indicator variables lists the values stored in indicator variables based on the values for fetched columns. The length of the value for the fetched column is L and the length of the receiving indicator variable is M.

[Table: Column values and values stored in indicator variables]



Data type of fetched column	**Indicator variable value**		
	Value of fetched column is a null value	**Value of fetched column is not a null value**	
		M < L	**M >= L**
NUMERIC DECIMAL INTEGER SMALLINT FLOAT REAL DOUBLE PRECISION DATE TIME TIMESTAMP INTERVAL BLOB	-1	0	0
CHARACTER (n) NATIONAL CHARACTER (n)	-1	n	0
CHARACTER VARYING NATIONAL CHARACTER VARYING	-1	Effective data size	0

Note: When a column value is a null value, the contents of the host variable in which data is stored are undefined.

If fetched data has a null value and no indicator variable is specified, processing for the single row SELECT statement ends in an error. If it is not known whether values for a column contain a null value, specify indicator variables.

2.1.3 Fetching data from multiple tables

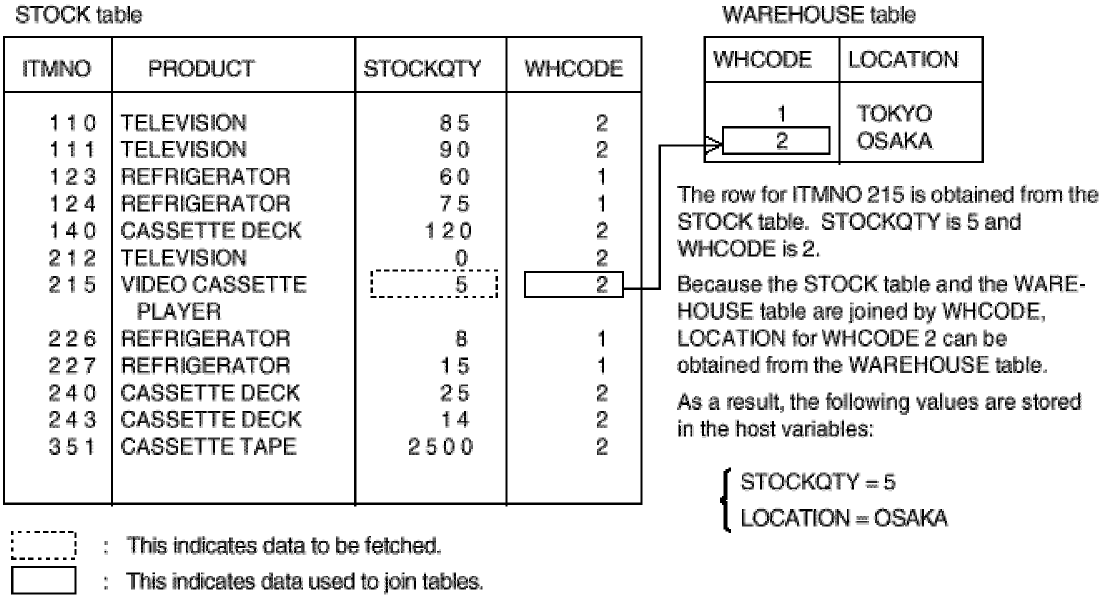
A single row SELECT statement can be used to join two or more tables to fetch data. An example of joining two tables follows. This section only explains the kinds of data that can be fetched as a result of joining tables. For information on the rules for joining and manipulating tables, see Chapter 5 "Joining Multiple Tables and Manipulating Data".

Example:

> In this example, STOCKQTY and warehouse location of the product with ITMNO "215" are obtained from the STOCK table. Data about the warehouse location is contained in the WAREHOUSE table. EQUIPMENT is the schema containing the WAREHOUSE table.

```
SELECT STOCKQTY,  LOCATION INTO  :STOCKQTY,  :LOCATION

      FROM STOCKS. STOCK, EQUIPMENT. WAREHOUSE

      WHERE ITMNO = 215 AND  STOCK. WHCODE = WAREHOUSE. WHCODE
```

[Figure: Example of joining two tables to fetch data]



STOCK table

ITMNO	PRODUCT	STOCKQTY	WHCODE
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
140	CASSETTE DECK	120	2
212	TELEVISION	0	2
215	VIDEO CASSETTE PLAYER	5	2
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETTE TAPE	2500	2

WAREHOUSE table

WHCODE	LOCATION
1	TOKYO
2	OSAKA

The row for ITMNO 215 is obtained from the STOCK table. STOCKQTY is 5 and WHCODE is 2.

Because the STOCK table and the WAREHOUSE table are joined by WHCODE, LOCATION for WHCODE 2 can be obtained from the WAREHOUSE table.

As a result, the following values are stored in the host variables:

STOCKQTY = 5
LOCATION = OSAKA

⬚ : This indicates data to be fetched.

☐ : This indicates data used to join tables.

2.1.4 Deleting duplicate rows and fetching data

A single row SELECT statement can be used to fetch only one row. An error occurs if a search condition specified in a single row SELECT statement is true for more than one row. An error also occurs if search conditions are not specified. If search conditions are not specified, data in all rows of the table is fetched. However, if when multiple rows are fetched the values in all rows for all columns fetched are equal, the result can be placed in one row and fetched. To do so, specify "DISTINCT". The following is an example of specifying DISTINCT:

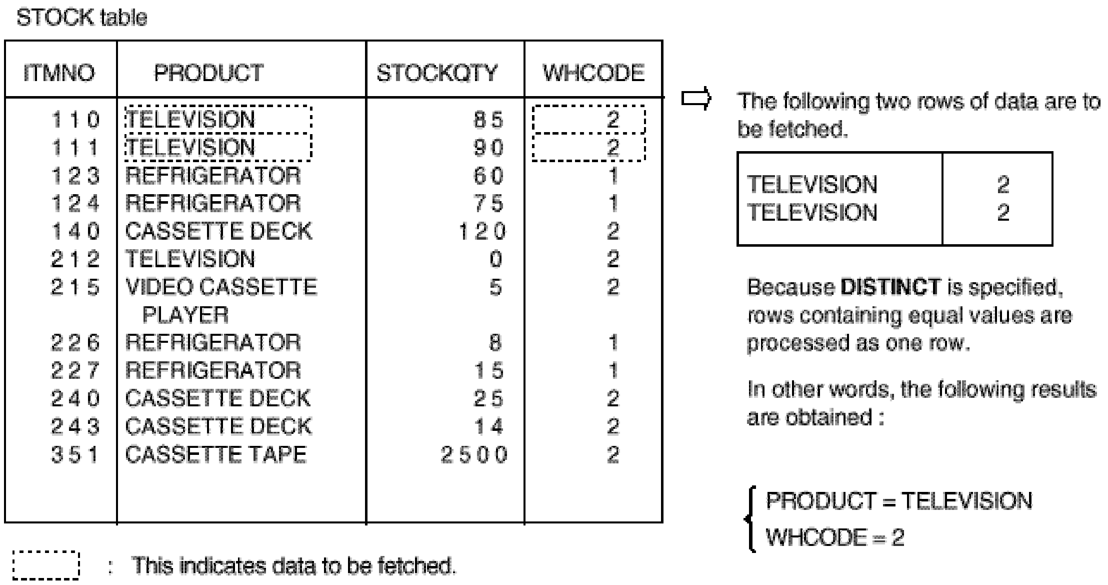
Example 1

> In this example, PRODUCT and WHCODE are obtained from the STOCK table for products for which ITMNO is less than "120". The rows for which these values are the same are processed as one row.

```
SELECT DISTINCT PRODUCT, WHCODE INTO  :PRODUCT,  :WHCODE
```

```
        FROM  STOCKS. STOCK WHERE  ITMNO < 120
```

[Figure: Example of deleting duplicate rows and fetching data]



STOCK table

ITMNO	PRODUCT	STOCKQTY	WHCODE
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
140	CASSETTE DECK	120	2
212	TELEVISION	0	2
215	VIDEO CASSETTE PLAYER	5	2
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETTE TAPE	2500	2

: This indicates data to be fetched.

The following two rows of data are to be fetched.

TELEVISION	2
TELEVISION	2

Because **DISTINCT** is specified, rows containing equal values are processed as one row.

In other words, the following results are obtained :

PRODUCT = TELEVISION
WHCODE = 2

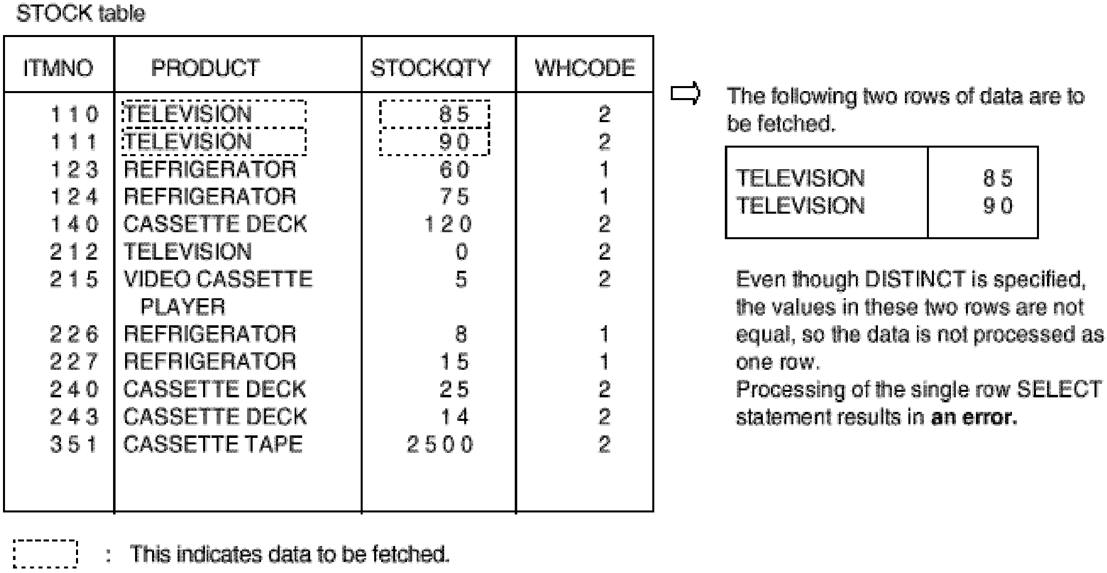
DISTINCT is used to form one row from fetched rows containing equal values. DISTINCT can easily be used incorrectly if the user does not realize that multiple rows will be fetched, which will result in an error. The following is an example of using DISTINCT incorrectly:

Example 2:

In this example, DISTINCT is used incorrectly. PRODUCT and STOCKQTY are obtained from the STOCK table for products for which ITMNO is less than "120".

```
SELECT DISTINCT  PRODUCT,  STOCKQTY INTO :PRODUCT,  :STOCKQTY

        FROM  STOCKS. STOCK WHERE  ITMNO < 120
```

[Figure: Example of specifying DISTINCT incorrectly]



STOCK table

ITMNO	PRODUCT	STOCKQTY	WHCODE
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
140	CASSETTE DECK	120	2
212	TELEVISION	0	2
215	VIDEO CASSETTE PLAYER	5	2
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETTE TAPE	2500	2

The following two rows of data are to be fetched.

TELEVISION	85
TELEVISION	90

Even though DISTINCT is specified, the values in these two rows are not equal, so the data is not processed as one row.
Processing of the single row SELECT statement results in **an error.**

: This indicates data to be fetched.

2.2 Fetching Columns that are Undefined

This section explains the following operations that can be performed by the single row SELECT statement:

- Performing arithmetic operations on data
- Obtaining the total, average, maximum, and minimum values, and the row count for column values

2.2.1 Performing arithmetic operations on data

So far, this manual has only explained the fetching of column values from tables without modification. However, a single row SELECT statement can also be used to fetch the results of arithmetic operations performed on column values. This is achieved by specifying an operational expression in a select column list. The following is an example of specifying an operational expression in a single row SELECT statement:

Example:

In this example, the result of adding 10 to STOCKQTY for the product with ITMNO "215" is fetched from the STOCK table.

```
SELECT  STOCKQTY + 10 INTO  :STOCKQTY


          FROM STOCKS. STOCK WHERE  ITMNO = 215
```

The "+" operator is used for addition, the "-" operator is used for subtraction, the "*" operator is used for multiplication, and the "/" operator is used for division. Also, a literal can be used to specify "10" as the value to be added to STOCKQTY. Operations can be performed on host variables and among columns as well as on columns and literals.

When the single row SELECT statement in the preceding example is executed, "15" is stored in host variable STOCKQTY.

■Specifying an operational expression

Monadic and dyadic operators can be used in operational expressions.

◆Monadic operators

The following are the two monadic operators:

+:

No effect

-:

Sign inversion

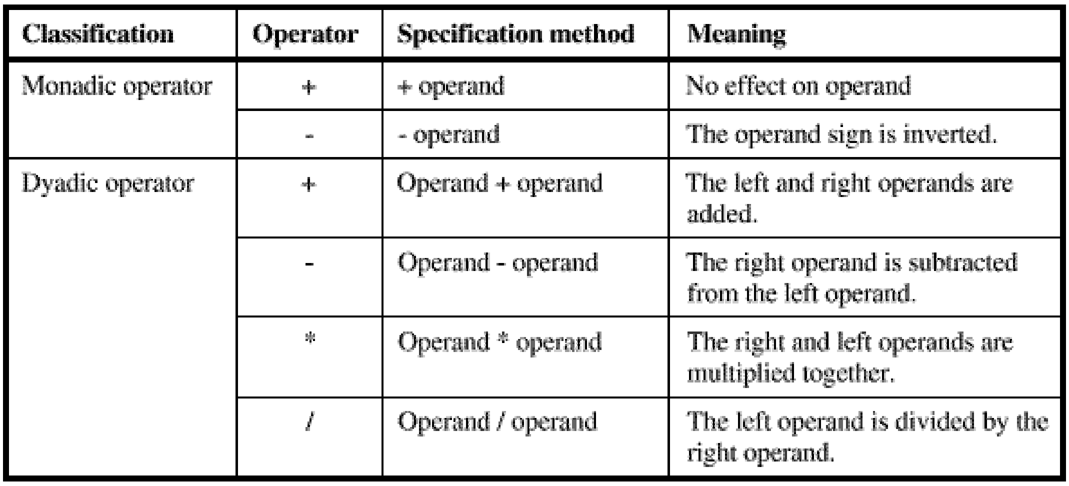
◆Dyadic operators

The following are the four dyadic operators:

+:
    Addition
-:
    Subtraction
*:
    Multiplication
/:
    Division

Table: Method of specifying and meaning of operational expressions shows the method of specifying, and the meaning of, operational expressions.

[Table: Method of specifying and meaning of operational expressions]



Classification	Operator	Specification method	Meaning
Monadic operator	+	+ operand	No effect on operand
	-	- operand	The operand sign is inverted.
Dyadic operator	+	Operand + operand	The left and right operands are added.
	-	Operand - operand	The right operand is subtracted from the left operand.
	*	Operand * operand	The right and left operands are multiplied together.
	/	Operand / operand	The left operand is divided by the right operand.

Specify a column name, host variable, or literal in the operator. Alternatively, the result of the operational expression can be specified as an operand. However, if an operational expression is specified in the operand of a monadic operator, the operational expression must be enclosed in parentheses.

■Priority of operations

The priority of operations is as follows:

1. Monadic operations
2. Multiplication and division
3. Addition and subtraction

Parentheses can also be used to specify priority. An expression within parentheses is executed before any other operation.

■Data types for results of operational expressions

This section explains the relationship between the data type for the result and the data type for the operand.

◆Data type for results of monadic operators

The data type for the results of monadic operators is the data type for the operand.

◆Data types for results of dyadic operators

Table: Data type for results of dyadic operators lists the data types for the results of dyadic operators. When more than one operator is specified in an operational expression, the data type for the results is determined for each operator. Follow the priority sequence for operators and apply the rules in Table: Data type for results of dyadic operators to all the operators in the operational expression to obtain the final data type for the operational expression.

[Table: Data type for results of dyadic operators]

Left operand	Right operand							
	SMALLINT	INTEGER	DECIMAL (p2, q2)	NUMERIC (p2, q2)	REAL	DOUBLE PRECISION	DATE TIME TIMESTAMP	INTERVAL
SMALLINT	INTEGER		[1]		[5]	[7]	error	INTERVAL(*5)
INTEGER			[2]					
DECIMAL (p1, q1)	[3] - Addition and subtraction DECIMAL (MIN(18, MAX(p1-q1, p2-q2) +MAX (q1, q2)+1), MAX(q1, q2) - Multiplication DECIMAL (MIN(18, p1+p2), q1+q2)(*2) - Division DECIMAL (MIN(18, p1-q1+q2+MAX (5, q1)), MAX (5, q1))	[4]	(*1)					
NUMERIC (p1, q1)								
REAL	[6]		REAL(*3)					
DOUBLE PRECISION	[8]		DOUBLE PRECISION(*4)					
DATE TIME TIMESTAMP	error						INTERVAL	DATE TIME TIMESTAMP
INTERVAL	INTERVAL(*6)						DATE TIME TIMESTAMP	INTERVAL

*1 If the left operand is SMALLINT, this is processed as p1 = 5, q1 = 0. ([1])

If the left operand is INTEGER, this is processed as p1 = 10, q1 = 0. ([2])

If the right operand is SMALLINT, this is processed as p2 = 5, q2 = 0. ([3])

If the right operand is INTEGER, this is processed as p2 = 10, q2 = 0. ([4])

*2 An error occurs if q1 + q2 > 18.

*3 If the left operand is SMALLINT, INTEGER, DECIMAL, or NUMERIC, this is processed as REAL. ([5])

If the right operand is SMALLINT, INTEGER, DECIMAL, or NUMERIC, this is processed as REAL. ([6])

*4 If the left operand is SMALLINT, INTEGER, DECIMAL, or NUMERIC, this is processed as DOUBLE PRECISION. ([7])

If the right operand is SMALLINT, INTEGER, DECIMAL, or NUMERIC, this is processed as DOUBLE PRECISION. ([8])

*5 An error occurs with dyadic operations other than multiplication.

*6 An error occurs with dyadic operations other than multiplication and division.

2.2.2 Obtaining total, average, maximum, and minimum value, and row count for column values

So far, this manual has only explained the fetching of data from required columns for one specified row from a table. The single row SELECT statement can be used to fetch the results of totaling data from multiple rows. By totaling the data, the total, average, maximum, and minimum values, and the row count for the data can be calculated. This is achieved by specifying a set function in the select column list. The following is an example of specifying a set function:

Example 1:

In this example, the total of ORDERQTY is fetched from the ORDER table.

```
SELECT SUM (ORDERQTY) INTO  :TOTALQTY FROM STOCKS. ORDER
```

Various set functions are available for determining values in columns. "SUM" is used to obtain the total value, "AVG" to obtain the average value, "MAX" to obtain the maximum value, "MIN" to obtain the minimum value, and "COUNT" to obtain the row count.

The reader has probably noticed that no WHERE clause was specified in the single row SELECT statement in Example 1 to specify the row to be fetched. The single row SELECT statement can be used to fetch data from only one row, so, in the previous examples in this manual, a condition specifying the row to be fetched was specified. However, in Example 1, since the total for all rows of a table was to be fetched, no condition specifying rows was required. Specify a WHERE clause to target only rows satisfying certain conditions for a total. When a WHERE clause is specified, only those rows for which the search condition is true are totaled.

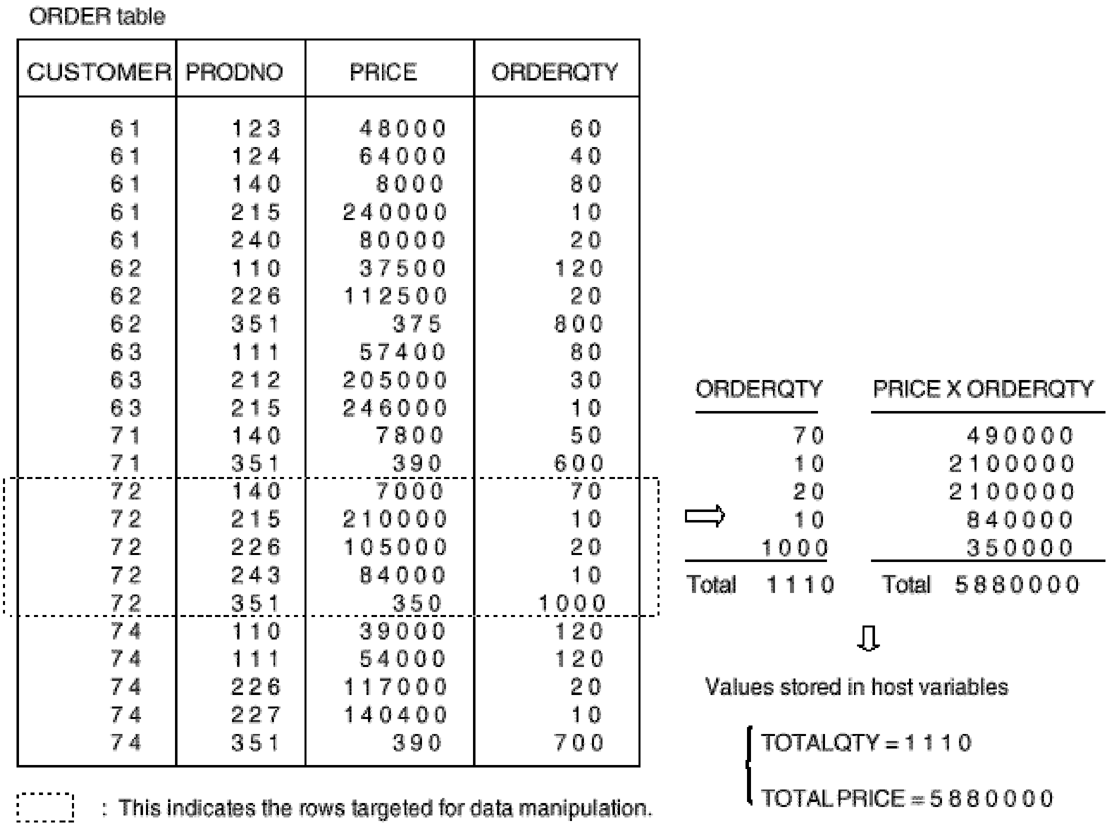
In Example 1, a set function was applied to the values in a column. Set functions can also be applied to the results of operational expressions. The following is an example of applying a set function to the results of an operational expression:

Example 2:

> In this example, the totals of ORDERQTY and order price are obtained from the ORDER table for CUSTOMER 72. Order price is obtained by multiplying PRICE by ORDERQTY.

```
SELECT SUM (ORDERQTY), SUM (PRICE * ORDERQTY)

       INTO :TOTALQTY, :TOTALPRICE

       FROM STOCKS. ORDER WHERE CUSTOMER = 72
```

[Figure: Example of applying a set function to the results of an operational expression]



ORDER table

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	123	48000	60
61	124	64000	40
61	140	8000	80
61	215	240000	10
61	240	80000	20
62	110	37500	120
62	226	112500	20
62	351	375	800
63	111	57400	80
63	212	205000	30
63	215	246000	10
71	140	7800	50
71	351	390	600
72	140	7000	70
72	215	210000	10
72	226	105000	20
72	243	84000	10
72	351	350	1000
74	110	39000	120
74	111	54000	120
74	226	117000	20
74	227	140400	10
74	351	390	700

ORDERQTY	PRICE X ORDERQTY
70	490000
10	2100000
20	2100000
10	840000
1000	350000
Total 1110	Total 5880000

Values stored in host variables

TOTALQTY = 1110

TOTAL PRICE = 5880000

⬚ : This indicates the rows targeted for data manipulation.

■Types of set functions and how to specify them

The following types of set function exist:

COUNT (*) function:
: Obtains the table row count including rows that contain null values

AVG function:
: Obtains average column value

MAX function:
: Obtains maximum column value

MIN function:
: Obtains minimum column value

SUM function:
: Obtains total column values

COUNT function:
: Obtains the table row count excluding rows that contain null values

Except for the COUNT (*) function, set functions can be specified in two ways. Note that both methods of specifying the MAX or MIN function produce the same result.

ALL:
: The set function is only applied to rows from the specified columns that do not contain null values.

DISTINCT:
: The set function is applied to rows from the specified column that do not contain null or duplicate values.

Functions in which ALL is specified are called ALL set functions. Functions in which DISTINCT is specified are called DISTINCT set functions. The default set function is ALL.

Figure: Rows processed by ALL and DISTINCT set functions shows an example of the rows processed by the ALL and DISTINCT set functions.

[Figure: Rows processed by ALL and DISTINCT set functions]

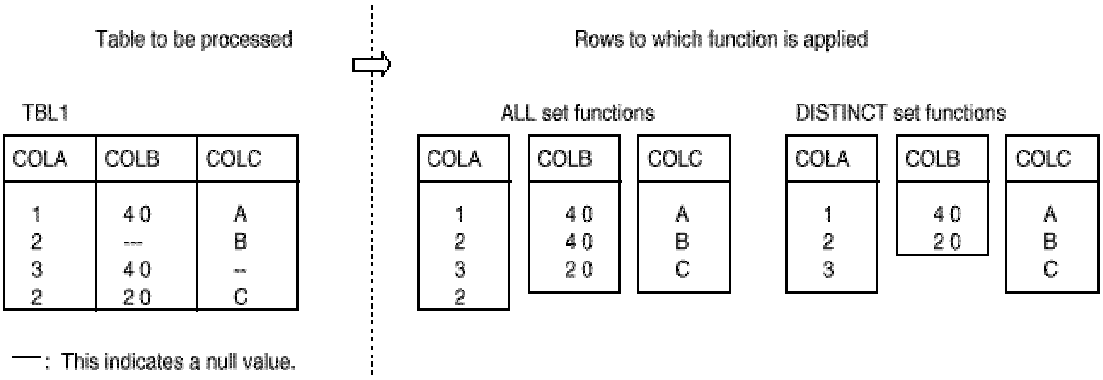


Table to be processed
Rows to which function is applied
TBL1
COLA COLB COLC
1 40 A
2 --- B
3 40 --
2 20 C
ALL set functions
COLA 1 2 3 2
COLB 40 40 20
COLC A B C
DISTINCT set functions
COLA 1 2 3
COLB 40 20
COLC A B C
—: This indicates a null value.

The methods of specifying set functions are listed in Table: Methods of specifying set functions.

[Table: Methods of specifying set functions]

Type	Function name	Specification method[*1]	Explanation
COUNT(*)		**COUNT(*)[*2]**	Table row count
ALL set function	AVG	**AVG (ALL value expression)**	Value expression average (Contains duplicate rows)
	MAX	**MAX (ALL value expression)**	Value expression maximum (Contains duplicate rows)
	MIN	**MIN (ALL value expression)**	Value expression minimum (Contains duplicate rows)
	SUM	**SUM (ALL value expression)**	DISTINCT set function (Contains duplicate rows)
	COUNT	**COUNT (ALL value expression)**	Value expression row count (Contains duplicate rows)
DISTINCT set function	AVG	**AVG (DISTINCT value expression)**	Value expression average (No duplicate rows)
	MAX	**MAX(DISTINCT value expression)**	Value expression maximum (No duplicate rows)
	MIN	**MIN(DISTINCT value expression)**	Value expression minimum (No duplicate rows)
	SUM	**SUM(DISTINCT value expression)**	Value expression total (No duplicate rows)
	COUNT	**COUNT(DISTINCT value expression)**	Value expression row count (No duplicate rows)

*1 When specifying an ALL set function, the keyword ALL can be omitted.

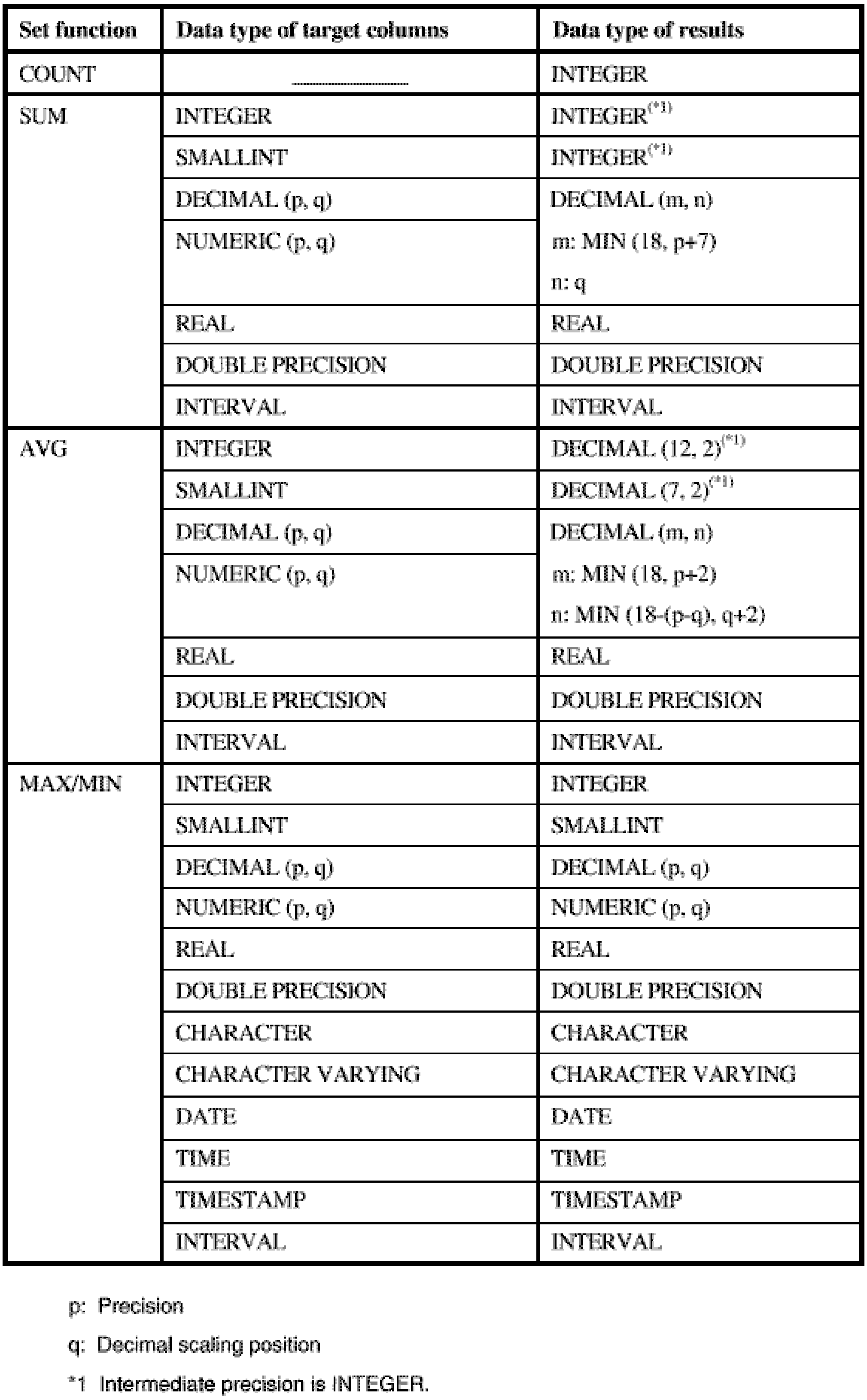
*2 Columns are not specified in COUNT (*).

Specify value expressions in arguments of ALL and DISTINCT set functions as shown in Table: Methods of specifying set functions. An operational expression that uses a column name can be specified in an argument.

■Data type of results of set functions

Table: Data type of columns targeted by set functions and data type of results shows the relationship between the data type of columns targeted by set functions and the data type of the results.

[Table: Data type of columns targeted by set functions and data type of the results]



Set function	Data type of target columns	Data type of results
COUNT	―――――	INTEGER
SUM	INTEGER	INTEGER(*1)
	SMALLINT	INTEGER(*1)
	DECIMAL (p, q)	DECIMAL (m, n) m: MIN (18, p+7) n: q
	NUMERIC (p, q)	
	REAL	REAL
	DOUBLE PRECISION	DOUBLE PRECISION
	INTERVAL	INTERVAL
AVG	INTEGER	DECIMAL (12, 2)(*1)
	SMALLINT	DECIMAL (7, 2)(*1)
	DECIMAL (p, q)	DECIMAL (m, n) m: MIN (18, p+2) n: MIN (18-(p-q), q+2)
	NUMERIC (p, q)	
	REAL	REAL
	DOUBLE PRECISION	DOUBLE PRECISION
	INTERVAL	INTERVAL
MAX/MIN	INTEGER	INTEGER
	SMALLINT	SMALLINT
	DECIMAL (p, q)	DECIMAL (p, q)
	NUMERIC (p, q)	NUMERIC (p, q)
	REAL	REAL
	DOUBLE PRECISION	DOUBLE PRECISION
	CHARACTER	CHARACTER
	CHARACTER VARYING	CHARACTER VARYING
	DATE	DATE
	TIME	TIME
	TIMESTAMP	TIMESTAMP
	INTERVAL	INTERVAL

p: Precision

q: Decimal scaling position

*1 Intermediate precision is INTEGER.

Chapter 3 Modifying Data

This chapter describes how to use the INSERT statement, UPDATE statement (searched) and DELETE statement (searched) to modify data.

This chapter contains the following sections:

3.1 Adding Data to a Data Base

3.2 Updating Data in a Data Base

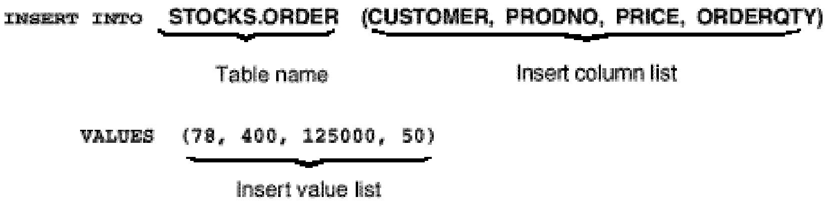
3.3 Deleting Data from a Data Base

3.1 Adding data to a Data Base

Use the INSERT statement to add data to a table. The following is an example of the INSERT statement:

Example:

In this example, the following data is added to the ORDER table: "78" for CUSTOMER, "400" for PRODNO, "125,000" for PRICE, and "50" for ORDERQTY.



```
INSERT INTO STOCKS.ORDER (CUSTOMER, PRODNO, PRICE, ORDERQTY)
    VALUES (78, 400, 125000, 50)
```

This chapter explains the following operations that can be performed by the INSERT statement:

- Specifying a null value in data to be added
- Using a default value in data to be added
- Specifying a current date and time in data to be added
- Adding data from another table

3.1.1 Specifying a null value in added data

To specify a null value in data to be added to a table, specify the keyword NULL or use an indicator variable instead of specifying a value in the insert value list. The following are examples of adding a null value in data:

Example 1:

In this example, the keyword NULL is specified. The following data is added to the ORDER table: "78" for CUSTOMER, "400" for PRODNO, and "50" for ORDERQTY. PRICE is specified as a null value.

```
INSERT INTO  STOCKS. ORDER  (CUSTOMER,  PRODNO,  PRICE,  ORDERQTY)

      VALUES (78, 400, NULL, 50)
```

Example 2:

In this example, an indicator variable is used. The data to be added is the same as in Example 1.

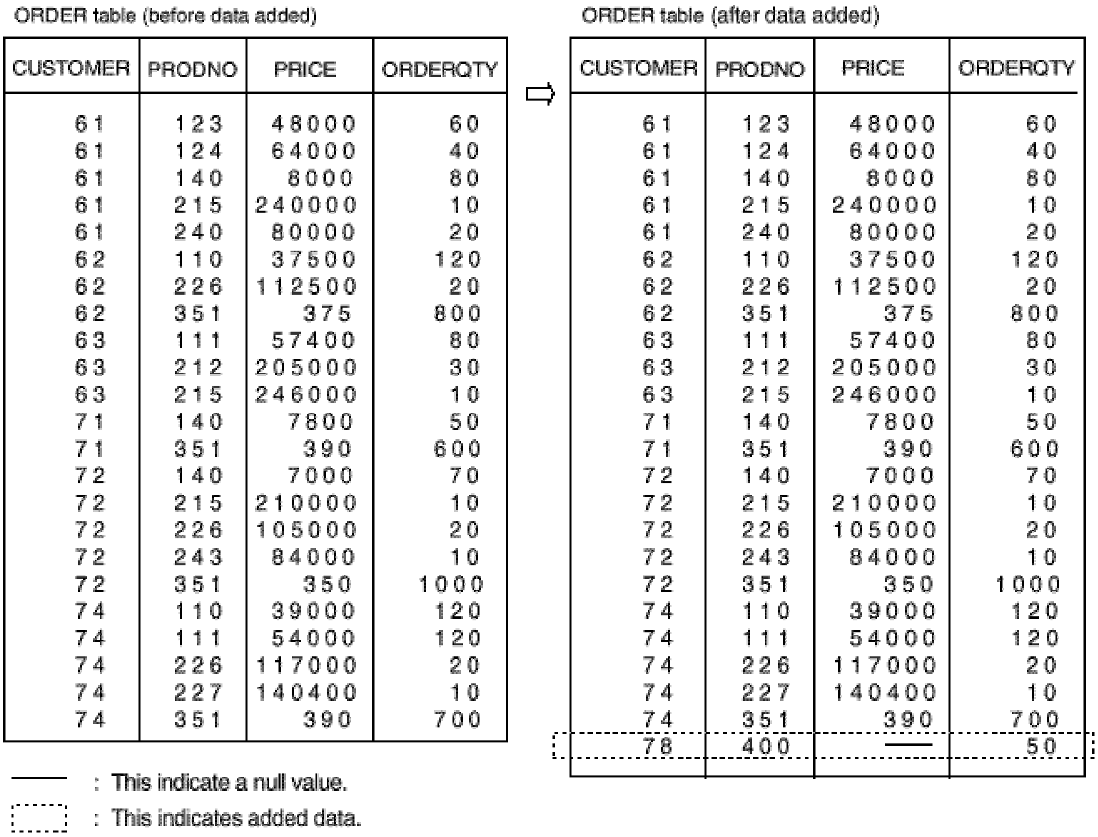
```
INSERT INTO STOCKS. ORDER  (CUSTOMER, PRODNO, PRICE, ORDERQTY)

      VALUES (78,  400,  :PRICE  INDICATOR  :PRICE_INDICATOR,  50)
```

In Example 2, "-1" is set as indicator variable PRICE_INDICATOR before the INSERT statement is executed. A value need not be set in host variable PRICE. In this case, the added data is a null value. However, if "0" or a positive value is specified as the indicator variable, then the data to be added becomes the value specified in host variable PRICE.

Therefore, if the data to be added contains a null value or some other value dependent on the processing of the application program, use an indicator variable. If an indicator variable is used, only one INSERT statement needs to be coded. However, for processing in which the data must be specified as a null value, specify the keyword "NULL" to simplify coding of the INSERT statement.

Figure: Example of adding a null value shows the execution results for the INSERT statement in Examples 1 and 2.

[Figure: Example of adding a null value]



ORDER table (before data added)

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	123	48000	60
61	124	64000	40
61	140	8000	80
61	215	240000	10
61	240	80000	20
62	110	37500	120
62	226	112500	20
62	351	375	800
63	111	57400	80
63	212	205000	30
63	215	246000	10
71	140	7800	50
71	351	390	600
72	140	7000	70
72	215	210000	10
72	226	105000	20
72	243	84000	10
72	351	350	1000
74	110	39000	120
74	111	54000	120
74	226	117000	20
74	227	140400	10
74	351	390	700

ORDER table (after data added)

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	123	48000	60
61	124	64000	40
61	140	8000	80
61	215	240000	10
61	240	80000	20
62	110	37500	120
62	226	112500	20
62	351	375	800
63	111	57400	80
63	212	205000	30
63	215	246000	10
71	140	7800	50
71	351	390	600
72	140	7000	70
72	215	210000	10
72	226	105000	20
72	243	84000	10
72	351	350	1000
74	110	39000	120
74	111	54000	120
74	226	117000	20
74	227	140400	10
74	351	390	700
78	400	——	50

—— : This indicate a null value.

┆┄┆ : This indicates added data.

Another method of specifying that data to be added is a null value is to omit the relevant column from the insert column list and the insert value list. The following is an example of omitting the relevant column:

Example 3:

> In this example, the column to which a null value is to be added is omitted. The data to be added is the same as in Example 1 and Example 2.

```
INSERT INTO STOCKS. ORDER  (CUSTOMER, PRODNO, ORDERQTY)


        VALUES (78, 400, 50)
```

In Example 3, PRICE is not specified as a column name in the insert column list, nor is a value corresponding to PRICE specified in the insert value list. In this case, a null value is defined as the default value for PRICE or the added data is a null value because a default value was not defined. For more information on adding data for which a default value is used, see "3.1.2 Using default values in data to be added."

Care must be taken to distinguish between whether the purpose of the processing for the application program is to add a null value or to add a default value. The default value for a column may have been changed due to changes in the design of the table. As a result, if the purpose of the processing is to add a null value, the methods shown in Examples 1 or 2 should be used.

3.1.2 Using default values in data to be added

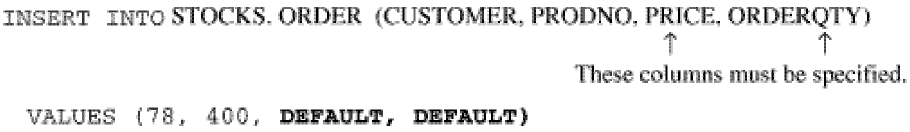
Default values, also referred to as fixed values, can be defined for each column in a table when the table is defined. The default values can be used by the INSERT statement to add data. If no value is specified for a column for which no default value is defined, a null value is added to the column. However, processing ends in an error if the column has a NOT NULL constraint.

■Setting a default value for a column in which data is to be added

To set a default value for a column in which data is to be added, specify the keyword "DEFAULT" in the insert value list of the INSERT statement. In this case, the name of the column for which the default value is to be set cannot be omitted from the insert column list. The following is an example in which default values for two columns are set:

Example 1:

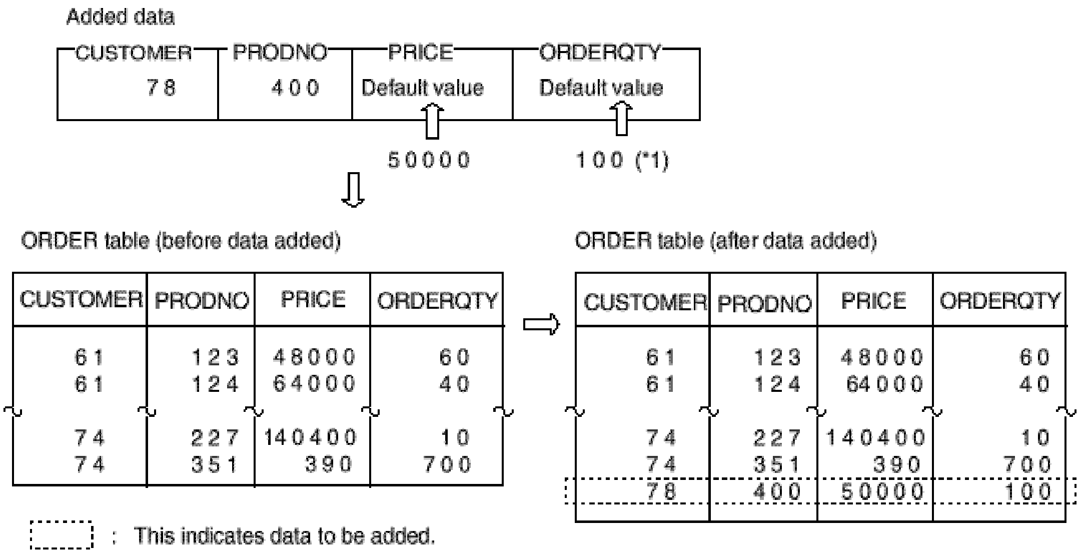
In this example, a row of data is added to the ORDER table. The data to be added is as follows: "78" for CUSTOMER and "400" for PRODNO. PRICE and ORDERQTY are specified with the default values defined for the columns.



```
INSERT INTO STOCKS.ORDER (CUSTOMER, PRODNO, PRICE, ORDERQTY)
 VALUES (78, 400, DEFAULT, DEFAULT)
```

↑ ↑
These columns must be specified.

[Figure: Adding data using default values]



Added data

CUSTOMER	PRODNO	PRICE	ORDERQTY
78	400	Default value	Default value

50000
100 (*1)
ORDER table (before data added)
ORDER table (after data added)

ORDER table (before data added)

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	123	48000	60
61	124	64000	40
~	~	~	~
74	227	140400	10
74	351	390	700

ORDER table (after data added)

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	123	48000	60
61	124	64000	40
~	~	~	~
74	227	140400	10
74	351	390	700
78	400	50000	100

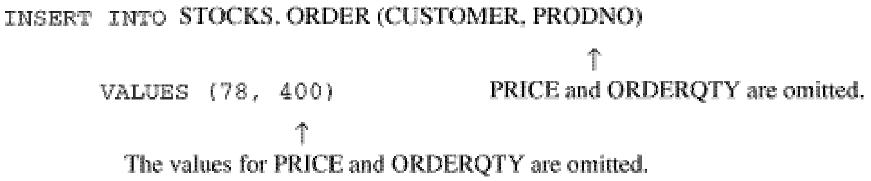
: This indicates data to be added.

*1 The default values defined are 50,000 for PRICE and 100 for ORDERQTY.

To set a default value for a column in which data is to be added, instead of specifying the keyword "DEFAULT", omit the value from the insert value list. In this case, also omit the name of the column for which the default value is to be set from the insert column list. The following is an example in which the insert value list and the insert column list for the INSERT statement in Example 1 are omitted:

Example 2:

In this example, the value to be added and the column to which it is to be added are omitted. The data added is the same as in Example 1.



```
INSERT  INTO STOCKS. ORDER (CUSTOMER, PRODNO)

             VALUES (78, 400)
```

PRICE and ORDERQTY are omitted.

The values for PRICE and ORDERQTY are omitted.

■Specifying null values for all columns in data to be added

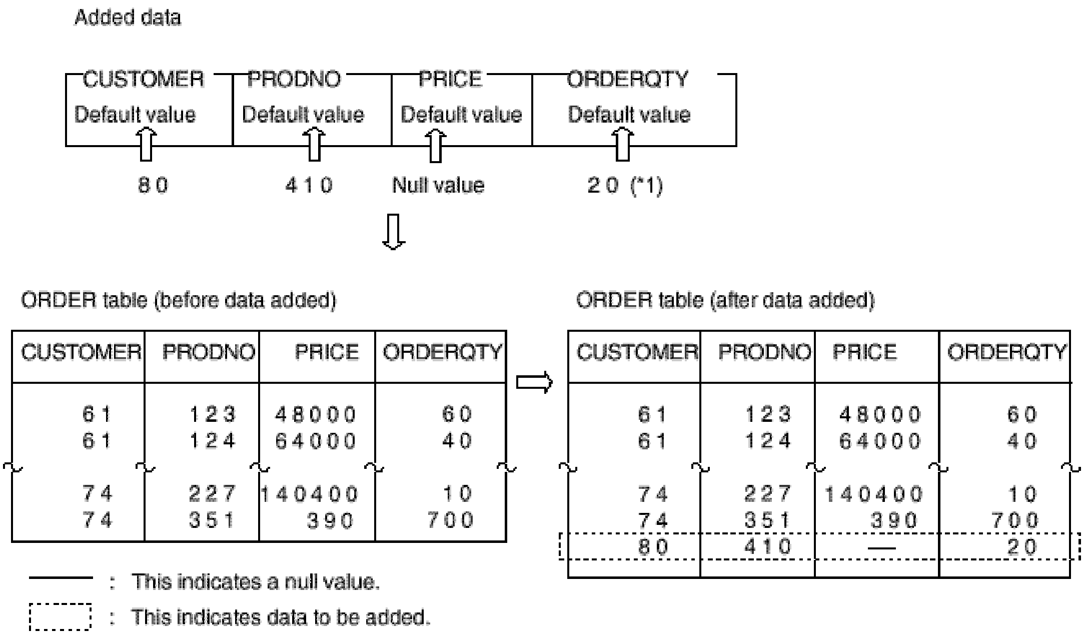
To set null values in all columns in data to be added, specify the keyword "DEFAULT VALUES" instead of specifying an insert column list or an insert value list. The following is an example in which DEFAULT VALUES is specified:

Example 3:

In this example, a row of data is added to the ORDER table. All the columns for the data to be added are specified with the default value defined for each column.

```
INSERT INTO  STOCKS. ORDER DEFAULT VALUES
```

[Figure: Adding data in which default values are used for all columns]



Added data
CUSTOMER
PRODNO
PRICE
ORDERQTY
Default value
Default value
Default value
Default value
80
410
Null value
20 (*1)
ORDER table (before data added)
ORDER table (after data added)
: This indicates a null value.
: This indicates data to be added.

ORDER table (before data added)

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	123	48000	60
61	124	64000	40
74	227	140400	10
74	351	390	700

ORDER table (after data added)

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	123	48000	60
61	124	64000	40
74	227	140400	10
74	351	390	700
80	410	—	20

*1 The default values defined are 80 for CUSTOMER, 410 for PRODNO, a null value for PRICE, and 20 for ORDERQTY.

3.1.3 Setting the current date and time in data to be added

To set the current date and time in the data to be added, specify the appropriate keyword in the insert value list of the INSERT statement. To set the current date, specify CURRENT_DATE. To set the current time, specify CURRENT_TIME. To set the current time-stamp, specify CURRENT_TIMESTAMP. An example of setting the current time in the data to be added is given below.

The example uses the DAILY_ORDER table, which contains the orders of a single day. The table consists of four columns: CUSTOMER, PRODNO, ORDERQTY, and PROCESS_TIME.

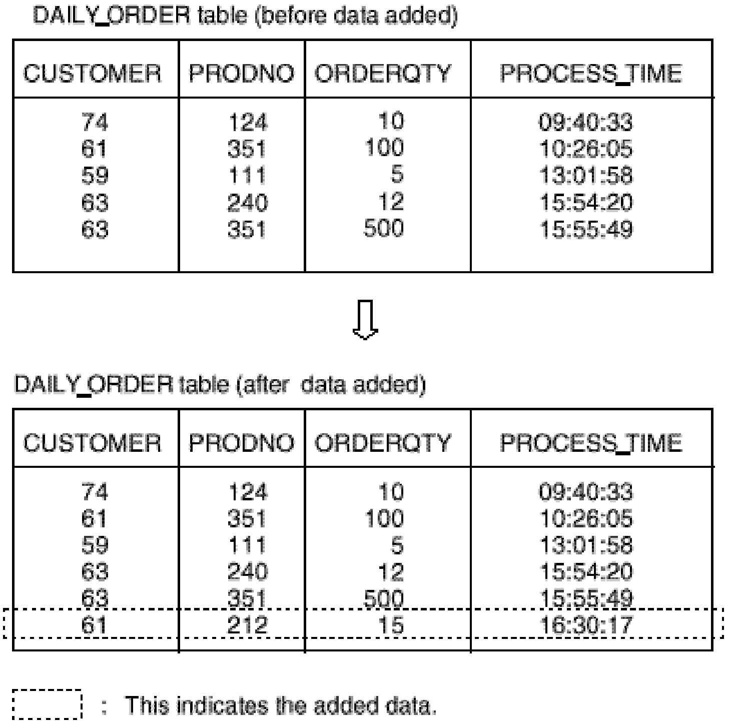
Example:

In this example, a row of data is added to the DAILY_ORDER table. The value "61" is specified for CUSTOMER, the value of host variable PRODNO is specified for PRODNO, and the value of host variable ORDERQTY is specified for ORDERQTY. The current time (16:30:17) is specified for

PROCESS_TIME. In this example, "212" has been specified for host variable PRODNO, and "15" has been specified for host variable ORDERQTY.

```
INSERT INTO STOCKS.DAILY_ORDER (CUSTOMER, PRODNO, ORDERQTY, PROCESS_TIME)

     VALUES (61, :PRODNO, :ORDERQTY, CURRENT_TIME)
```

[Figure: Adding data using the current time]



DAILY_ORDER table (before data added)

CUSTOMER	PRODNO	ORDERQTY	PROCESS_TIME
74	124	10	09:40:33
61	351	100	10:26:05
59	111	5	13:01:58
63	240	12	15:54:20
63	351	500	15:55:49

⇩

DAILY_ORDER table (after data added)

CUSTOMER	PRODNO	ORDERQTY	PROCESS_TIME
74	124	10	09:40:33
61	351	100	10:26:05
59	111	5	13:01:58
63	240	12	15:54:20
63	351	500	15:55:49
61	212	15	16:30:17

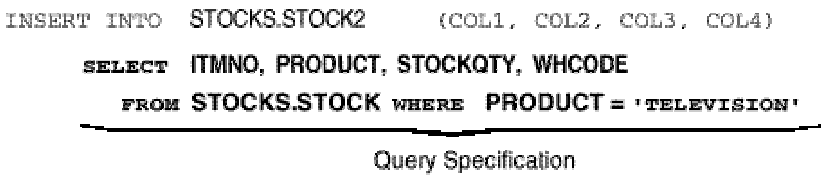
: This indicates the added data.

3.1.4 Adding data from another table

Depending on the processing performed by the application program, data may need to be fetched from one table and added to another. In such cases, processing is not very efficient if a single row SELECT statement or a cursor is used to fetch and add data. To improve efficiency, an INSERT statement can be used to add data from one table to another table. Use a query specification to fetch data to be added from a table. The following is an example of a query specification:

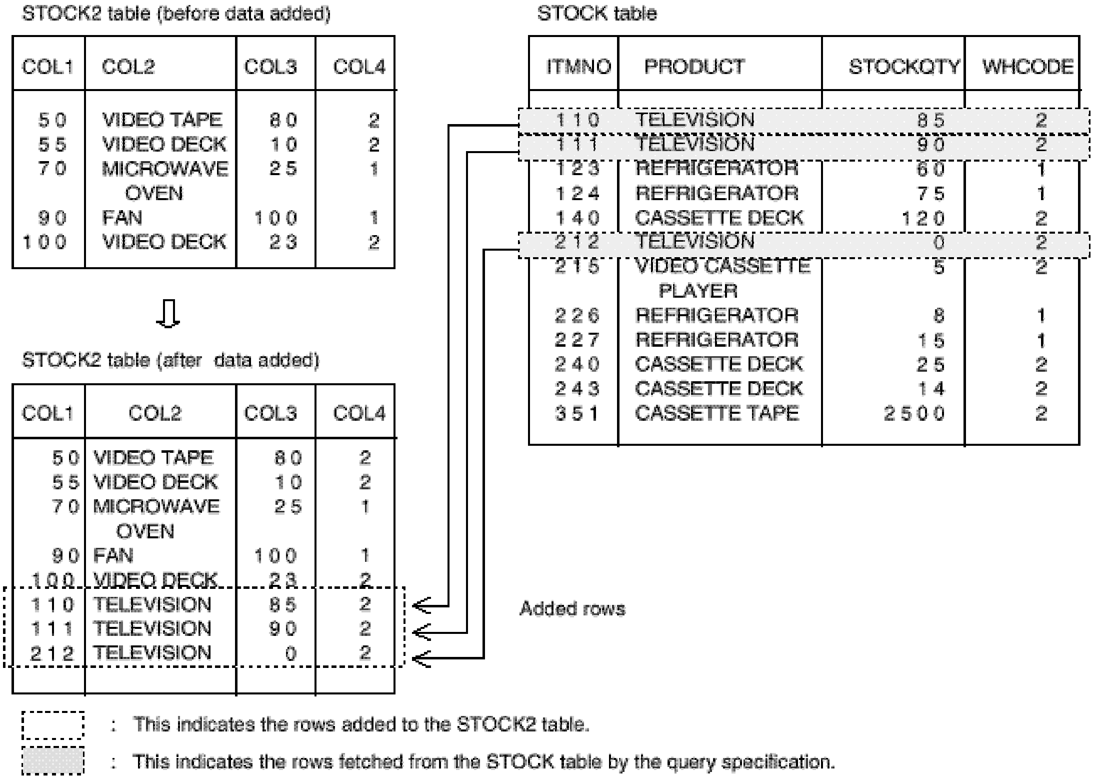
Example 1:

In this example, the row for PRODUCT TELEVISION is added to table STOCK2 table from the STOCK table.



```
INSERT INTO STOCKS.STOCK2     (COL1, COL2, COL3, COL4)
      SELECT ITMNO, PRODUCT, STOCKQTY, WHCODE
         FROM STOCKS.STOCK WHERE PRODUCT = 'TELEVISION'
```

[Figure: Using a query specification to add data (where the column layout is the same)]



STOCK2 table (before data added)

COL1	COL2	COL3	COL4
50	VIDEO TAPE	80	2
55	VIDEO DECK	10	2
70	MICROWAVE OVEN	25	1
90	FAN	100	1
100	VIDEO DECK	23	2

STOCK2 table (after data added)

COL1	COL2	COL3	COL4
50	VIDEO TAPE	80	2
55	VIDEO DECK	10	2
70	MICROWAVE OVEN	25	1
90	FAN	100	1
100	VIDEO DECK	23	2
110	TELEVISION	85	2
111	TELEVISION	90	2
212	TELEVISION	0	2

STOCK table

ITMNO	PRODUCT	STOCKQTY	WHCODE
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
140	CASSETTE DECK	120	2
212	TELEVISION	0	2
215	VIDEO CASSETTE PLAYER	5	2
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETTE TAPE	2500	2

Added rows
: This indicates the rows added to the STOCK2 table.
: This indicates the rows fetched from the STOCK table by the query specification.

Note: The data in the STOCK table is not changed.

In Example 1, the layout of the columns in the table to which data is added and the table from which data is fetched have a one-to-one correspondence. Usually, the layout of columns in tables does not correspond so closely. For example, the data from some columns in a table may be added to another table. Or, a value determined beforehand by an application program may be added to a column in the table to which data is to be added. An example of this follows.

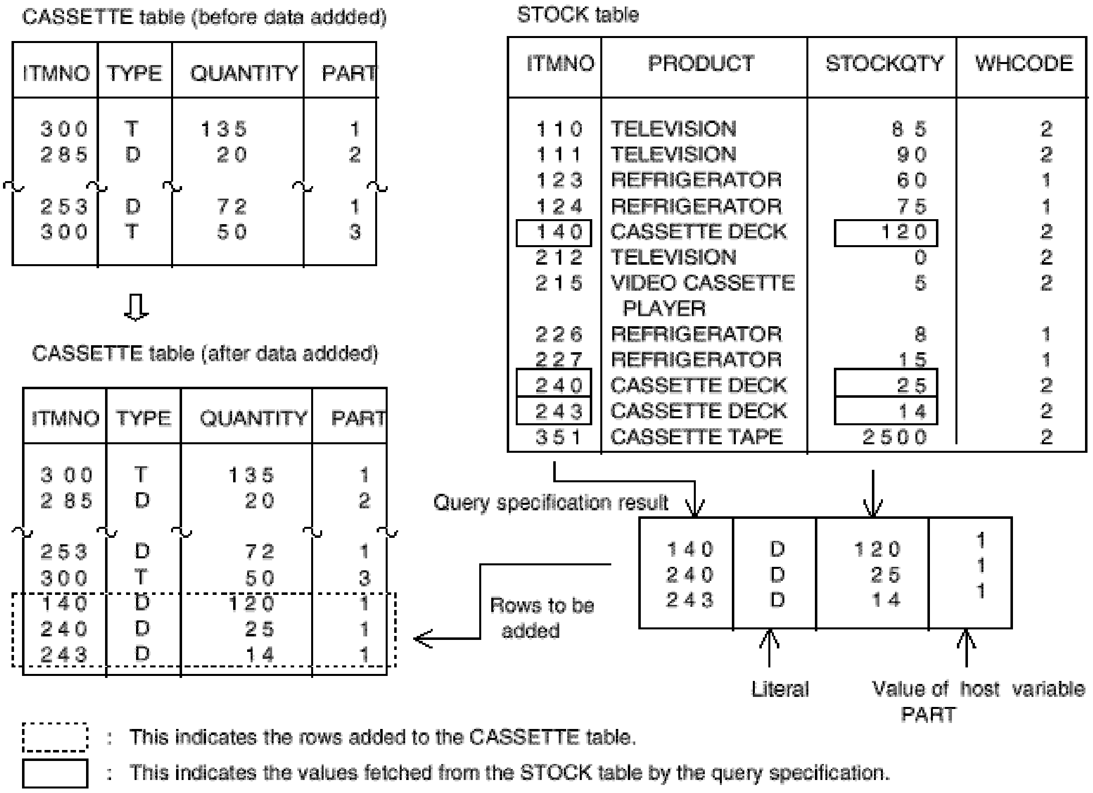
Suppose a table named CASSETTE table containing STOCKQTY and ORDERQTY data for CASSETTE DECK and CASSETTE TAPE. This table consists of four columns: ITMNO, TYPE, QUANTITY, and PART. CASSETTE DECK is set as "D" and CASSETTE TAPE as "T" in column TYPE. The quantity of stock and orders is set in column QUANTITY. In column PART, stock is "1", orders placed with CUSTOMER is "2", and orders placed with multiple CUSTOMER is "3".

Example 2:

In this example, ITMNO and STOCKQTY from the STOCK table are added to the CASSETTE table. Also, "D", which stands for CASSETTE DECK, is set in column TYPE, and "1", which stands for stock, is set in host variable PART.

```
INSERT  INTO STOCKS. CASSETTE  (ITMNO,  TYPE,  QUANTITY,  PART)
     SELECT  ITMNO,  'D',  STOCKQTY,  :PART
          FROM STOCKS. STOCK WHERE PRODUCT = 'CASSETTE DECK'
```

[Figure: Using a query specification to add data (when column layout does not correspond)]

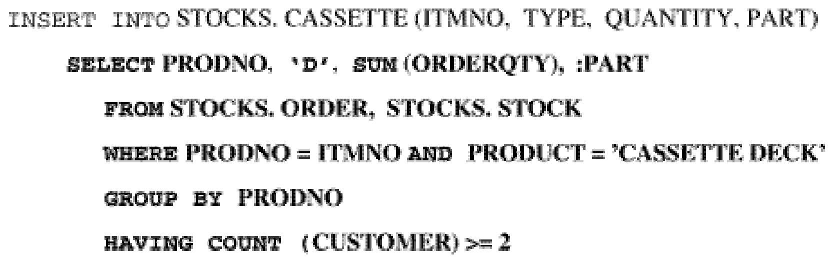

CASSETTE table (before data addded)
ITMNO
TYPE
QUANTITY
PART
CASSETTE table (after data addded)
STOCK table
ITMNO
PRODUCT
STOCKQTY
WHCODE
Query specification result
Rows to be added
Literal
Value of host variable PART
: This indicates the rows added to the CASSETTE table.
: This indicates the values fetched from the STOCK table by the query specification.

Note: The data in the STOCK table is not changed.

Besides fetching specific values from a table, query specifications can also be used to join and manipulate multiple tables. Query specifications can also be used to group tables and total their values. An example of this follows. For detailed information, see Chapter 5 "Joining Multiple tables and Manipulating Data" and 6.1 "Grouping tables and Manipulating Data". Do not be concerned too much about the details of Figure: Adding data using a complex query specification, just try to understand in general terms what happens when a query specification is used.

Example 3:

In this example, PRODNO for the ORDER table and the total of ORDERQTY for each PRODNO is added to the CASSETTE table. The data to be added is specified as only for orders by more than one CUSTOMER. In addition, "D", which stands for CASSETTE DECK, is set in column TYPE, and "3", which stands for orders by more than one CUSTOMER, is set in host variable PART.


```
INSERT  INTO STOCKS. CASSETTE (ITMNO,  TYPE,  QUANTITY, PART)
    SELECT PRODNO,  'D',  SUM (ORDERQTY),  :PART
      FROM STOCKS. ORDER,  STOCKS. STOCK
      WHERE PRODNO = ITMNO AND  PRODUCT = 'CASSETTE DECK'
      GROUP  BY  PRODNO
      HAVING  COUNT  (CUSTOMER) >= 2
```

[Figure: Adding data using a complex query specification]

Table derived from the FROM clause

CUSTOMER	PRODNO	PRICE	ORDERQTY	ITMNO	PRODUCT	STOCKQTY	WHCODE
61	123	48000	60	110	TELEVISION	85	2
61	123	48000	60	111	TELEVISION	90	2
61	123	48000	60	123	REFRIGERATOR	60	1
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.

(The table derived from the FROM clause contains the 276 rows of the ORDER table and STOCK table.)

⇩

Table derived from the WHERE clause

CUSTOMER	PRODNO	PRICE	ORDERQTY	ITMNO	PRODUCT	STOCKQTY	WHCODE
61	140	8000	80	140	CASSETTE DECK	120	2
61	240	80000	20	240	CASSETTE DECK	25	2
71	140	7800	50	140	CASSETTE DECK	120	2
72	140	7000	70	140	CASSETTE DECK	120	2
72	243	84000	10	243	CASSETTE DECK	14	2

(The table derived from the WHERE clause contains five rows. These are the rows in which PRODUCT is CASSETTE DECK and in which the values of PRODNO and ITMNO are equal in the table derived from the FROM clause.)

⇩

Table derived from the GROUP BY clause

CUSTOMER	PRODNO	PRICE	ORDERQTY	ITMNO	PRODUCT	STOCKQTY	WHCODE
61	140	8000	80	140	CASSETTE DECK	120	2
71	140	7800	50	140	CASSETTE DECK	120	2
72	140	7000	70	140	CASSETTE DECK	120	2
61	240	80000	20	240	CASSETTE DECK	25	2
72	243	84000	10	243	CASSETTE DECK	14	2

⇩

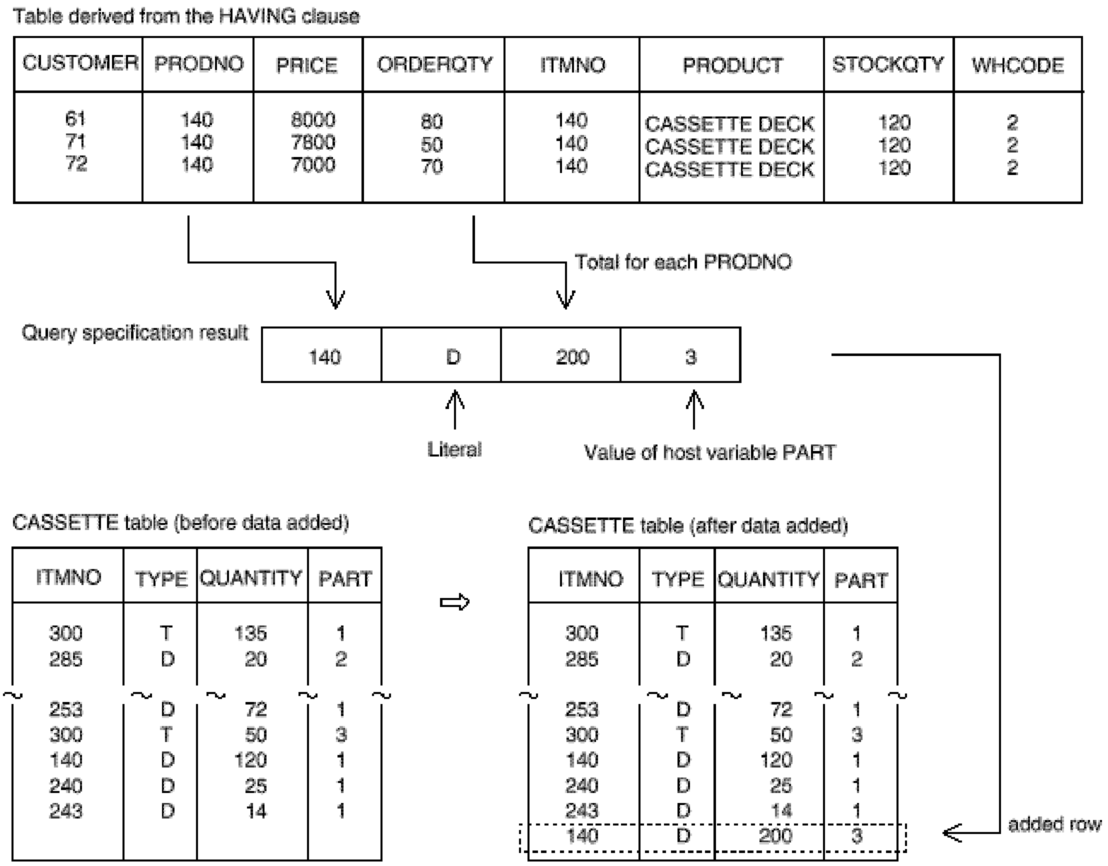


Table derived from the HAVING clause
Total for each PRODNO
Query specification result
Literal
Value of host variable PART
CASSETTE table (before data added)
CASSETTE table (after data added)
added row

Table derived from the HAVING clause

CUSTOMER	PRODNO	PRICE	ORDERQTY	ITMNO	PRODUCT	STOCKQTY	WHCODE
61	140	8000	80	140	CASSETTE DECK	120	2
71	140	7800	50	140	CASSETTE DECK	120	2
72	140	7000	70	140	CASSETTE DECK	120	2

Query specification result

140	D	200	3

CASSETTE table (before data added)

ITMNO	TYPE	QUANTITY	PART
300	T	135	1
285	D	20	2
253	D	72	1
300	T	50	3
140	D	120	1
240	D	25	1
243	D	14	1

CASSETTE table (after data added)

ITMNO	TYPE	QUANTITY	PART
300	T	135	1
285	D	20	2
253	D	72	1
300	T	50	3
140	D	120	1
240	D	25	1
243	D	14	1
140	D	200	3

┆┄┄┆ : This indicates the rows added to the CASSETTE table.

Note: The data in the ORDER table and STOCK table is not changed.

3.2 Updating data in a Data Base

Use the UPDATE statement to update data in a table. The following is an example of the UPDATE statement:

Example:

In this example, ORDERQTY is updated to "50" for the data for PRODNO "215" and CUSTOMER "61" in the ORDER table.

$$\text{UPDATE } \underbrace{\text{STOCKS.ORDER}}_{\text{Table name}} \text{ SET } \underbrace{\text{ORDERQTY = 50}}_{\text{Set clause}}$$

$$\text{WHERE } \underbrace{\text{CUSTOMER = 61 AND PRODNO = 215}}_{\text{Search condition}}$$

This section explains the following operations that can be performed by the UPDATE statement:

- Updating values in multiple columns
- Specifying a null value for data to be updated
- Using default values in data to be updated
- Specifying the current date and time in data to be updated
- Using values from other columns in data to be updated
- Performing arithmetic operations on data to be updated
- Updating all rows in a table

Data can also be updated using a cursor. For information on this method, see 4.5 "Updating Data Using a Cursor".

The UPDATE statement explained in this section is called the UPDATE statement (searched) because search conditions are used to specify rows to be updated.

3.2.1 Updating values in multiple columns

In the first example in 3.2 "Updating Data in a Data Base", data was updated in column ORDERQTY only. Suppose that data in two columns, PRICE and ORDERQTY, is to be updated. An UPDATE statement could be executed once for each column, but this is not efficient. Instead, the UPDATE statement can update values in multiple columns at the same time.

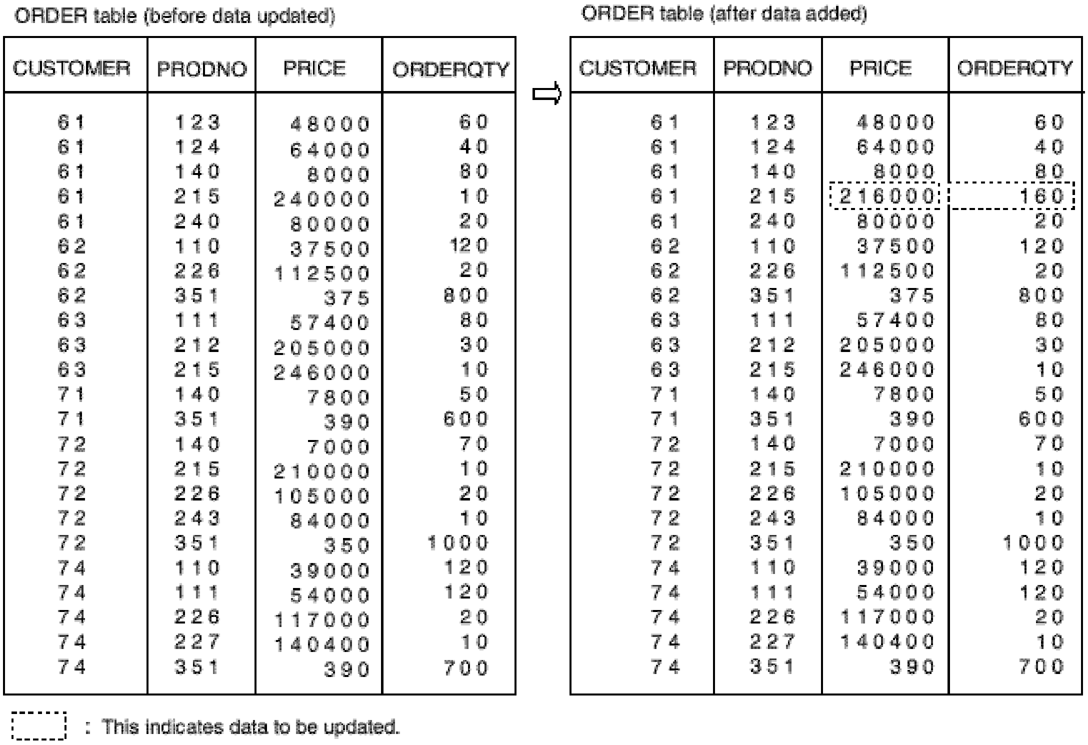
To update data in multiple columns, specify the columns by separating each set clause with a comma. The values for columns not specified in the set clause are not changed. The following is an example of updating multiple columns:

Example:

> In this example, PRICE is updated to "216,000" and ORDERQTY is updated to "160" for the data for CUSTOMER "61" and PRODNO "215" in the ORDER table.

```
UPDATE STOCKS. ORDER SET PRICE = 216000,  ORDERQTY = 160

     WHERE CUSTOMER = 61 AND  PRODNO = 215
```

[Figure: Example of updating multiple columns]



ORDER table (before data updated)

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	123	48000	60
61	124	64000	40
61	140	8000	80
61	215	240000	10
61	240	80000	20
62	110	37500	120
62	226	112500	20
62	351	375	800
63	111	57400	80
63	212	205000	30
63	215	246000	10
71	140	7800	50
71	351	390	600
72	140	7000	70
72	215	210000	10
72	226	105000	20
72	243	84000	10
72	351	350	1000
74	110	39000	120
74	111	54000	120
74	226	117000	20
74	227	140400	10
74	351	390	700

⇨

ORDER table (after data added)

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	123	48000	60
61	124	64000	40
61	140	8000	80
61	215	216000	160
61	240	80000	20
62	110	37500	120
62	226	112500	20
62	351	375	800
63	111	57400	80
63	212	205000	30
63	215	246000	10
71	140	7800	50
71	351	390	600
72	140	7000	70
72	215	210000	10
72	226	105000	20
72	243	84000	10
72	351	350	1000
74	110	39000	120
74	111	54000	120
74	226	117000	20
74	227	140400	10
74	351	390	700

⬚ : This indicates data to be updated.

3.2.2 Specifying a null value for data to be updated

To update data in a table to a null value, specify the keyword "NULL" instead of specifying a value in the set clause or use an indicator variable. The following is an example of updating data to a null value:

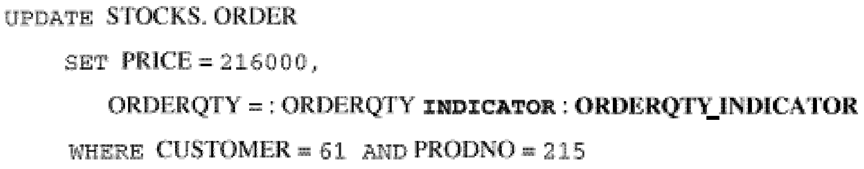
Example 1:

> In this example, the keyword "NULL" is specified. PRICE is updated to "216,000" and ORDERQTY is updated to a null value for data for CUSTOMER "61" and PRODNO "215" in the ORDER table.

```
UPDATE  STOCKS. ORDER SET PRICE = 216000,  ORDERQTY = NULL


         WHERE  CUSTOMER = 61 AND PRODNO = 215
```

Example 2:

In this example, an indicator variable is used. The data to be updated is the same as in Example 1.



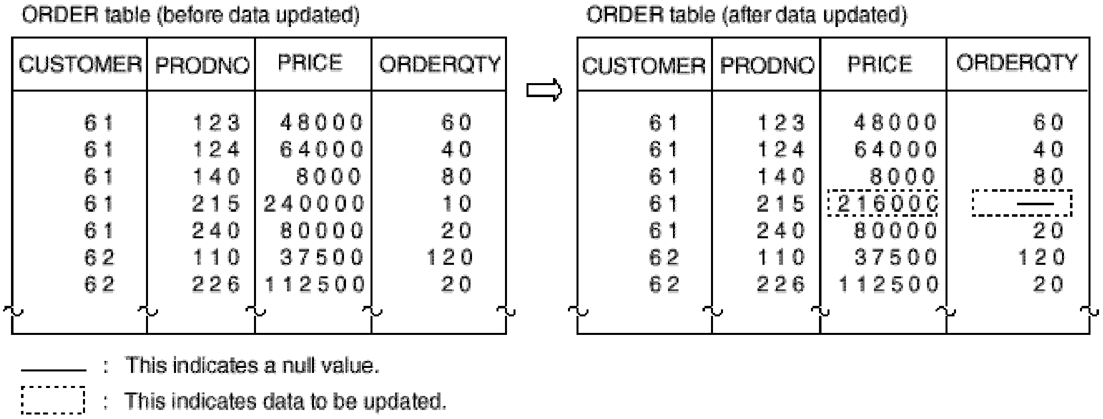
```
UPDATE STOCKS. ORDER
      SET PRICE = 216000,
          ORDERQTY = : ORDERQTY INDICATOR : ORDERQTY_INDICATOR
      WHERE CUSTOMER = 61  AND PRODNO = 215
```

In Example 2, "-1" is set for indicator variable ORDERQTY_INDICATOR, then the UPDATE statement is executed. A value need not be specified in host variable ORDERQTY. In this case, the data to be updated becomes a null value. If "0" or a positive value is set for the indicator variable, the data to be updated becomes the value specified in host variable ORDERQTY.

Therefore, if the data to be updated is a null value or some other value depending on the processing for the application program, use an indicator variable. If an indicator variable is used, only one UPDATE statement needs to be coded. However, in processing in which the data must be specified as a null value, specify the keyword "NULL" to simplify the coding of the UPDATE statement.

Figure: Example in which data is updated to a null value shows the execution results for the UPDATE statements in Examples 1 and 2.

[Figure: Example in which data is updated to a null value]



ORDER table (before data updated)

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	123	48000	60
61	124	64000	40
61	140	8000	80
61	215	240000	10
61	240	80000	20
62	110	37500	120
62	226	112500	20

ORDER table (after data updated)

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	123	48000	60
61	124	64000	40
61	140	8000	80
61	215	216000	—
61	240	80000	20
62	110	37500	120
62	226	112500	20

——— : This indicates a null value.

⬚ : This indicates data to be updated.

3.2.3 Using default values in data to be updated

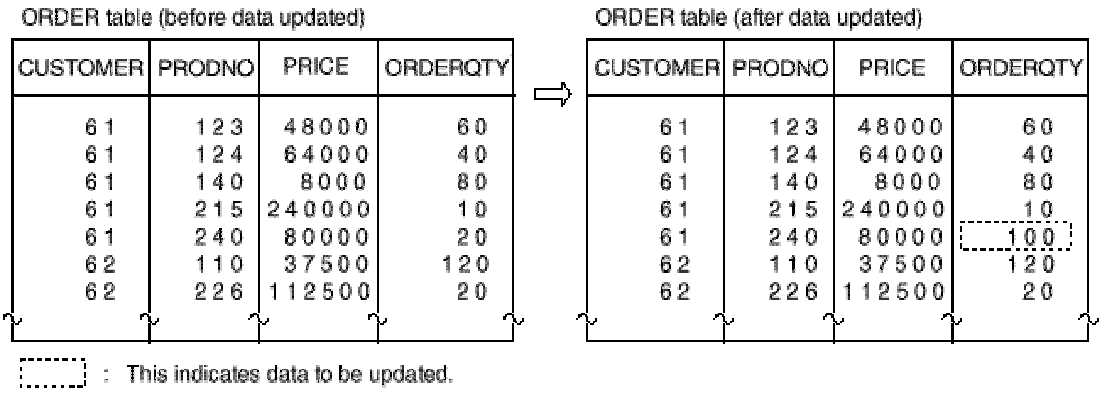
Adding data using default values is explained in 3.1.2 "Using default values in data to be added". The UPDATE statement can also be used to update data using default values. To set a default value in data to be updated, specify the keyword DEFAULT instead of specifying values in a set clause. If no default value is defined for the column specified in the set clause, the data in the column is updated to a null value. However, this processing ends in an error if a column with a NOT NULL constraint exists. The following is an example of specifying the keyword DEFAULT:

Example:

In this example, the keyword "DEFAULT" is specified. ORDERQTY is updated to the default value for the data for PRODNO "240" in the ORDER table.

```
UPDATE  STOCKS. ORDER  SET  ORDERQTY = DEFAULT  WHERE  PRODNO = 240
```

[Figure: Updating data using a default value]



ORDER table (before data updated)

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	123	48000	60
61	124	64000	40
61	140	8000	80
61	215	240000	10
61	240	80000	20
62	110	37500	120
62	226	112500	20

ORDER table (after data updated)

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	123	48000	60
61	124	64000	40
61	140	8000	80
61	215	240000	10
61	240	80000	100
62	110	37500	120
62	226	112500	20

: This indicates data to be updated.

Note: The default value defined for ORDERQTY is 100.

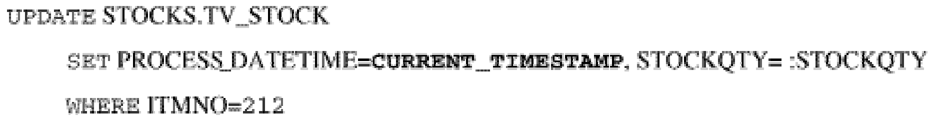
3.2.4 Specifying the current date and time in data to be updated

To specify the current date and time in data to be added, specify the appropriate keyword in the set clause instead of a value. To specify the current date, specify CURRENT_DATE. To specify the current time, specify CURRENT_TIME. To specify the current time-stamp, specify CURRENT_TIMESTAMP. An example of specifying the current time-stamp in data to be updated is given below.

The example uses the TV_STOCK table, which consists of three columns: PROCESS_DATETIME, ITMNO, and STOCKQTY. STOCKQTY at PROCESS_DATETIME is specified for STOCKQTY.

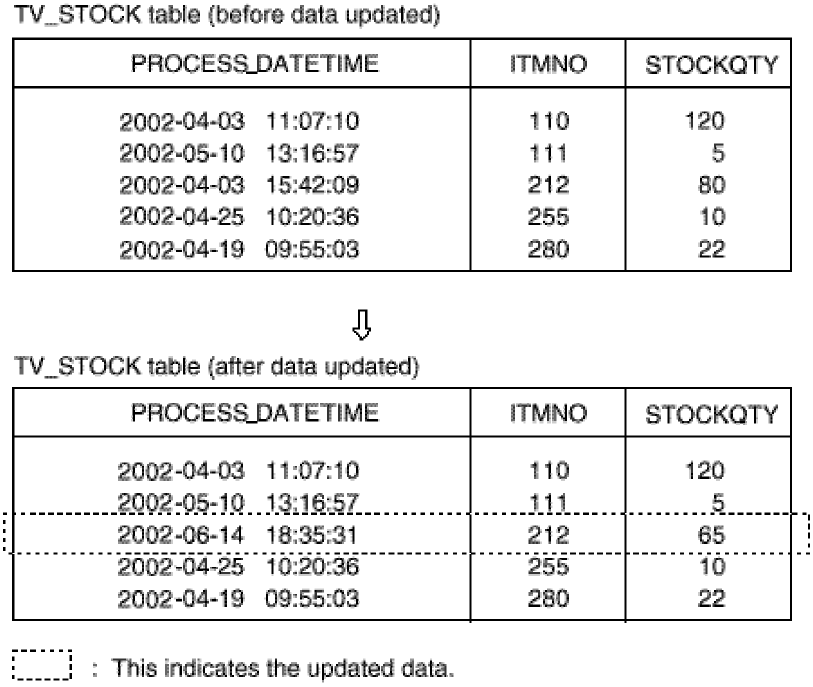
Example:

> In this example, PROCESS_DATETIME is updated to the current date and time (October 14, 2000, 18:35:31) and STOCKQTY is updated to the value of host variable STOCKQTY. PROCESS_DATETIME and STOCKQTY are updated for the data for ITMNO "212" in the TV_STOCK table. The value "65" has been specified for host variable STOCKQTY.



```
UPDATE STOCKS.TV_STOCK
    SET PROCESS_DATETIME=CURRENT_TIMESTAMP, STOCKQTY= :STOCKQTY
    WHERE ITMNO=212
```

[Figure: Updating data using the current date and time]



TV_STOCK table (before data updated)

PROCESS_DATETIME	ITMNO	STOCKQTY
2002-04-03 11:07:10	110	120
2002-05-10 13:16:57	111	5
2002-04-03 15:42:09	212	80
2002-04-25 10:20:36	255	10
2002-04-19 09:55:03	280	22

⇩

TV_STOCK table (after data updated)

PROCESS_DATETIME	ITMNO	STOCKQTY
2002-04-03 11:07:10	110	120
2002-05-10 13:16:57	111	5
2002-06-14 18:35:31	212	65
2002-04-25 10:20:36	255	10
2002-04-19 09:55:03	280	22

⬚ : This indicates the updated data.

3.2.5 Using values from other columns in data to be updated

Values from other columns can be used to update data. An example of this follows.

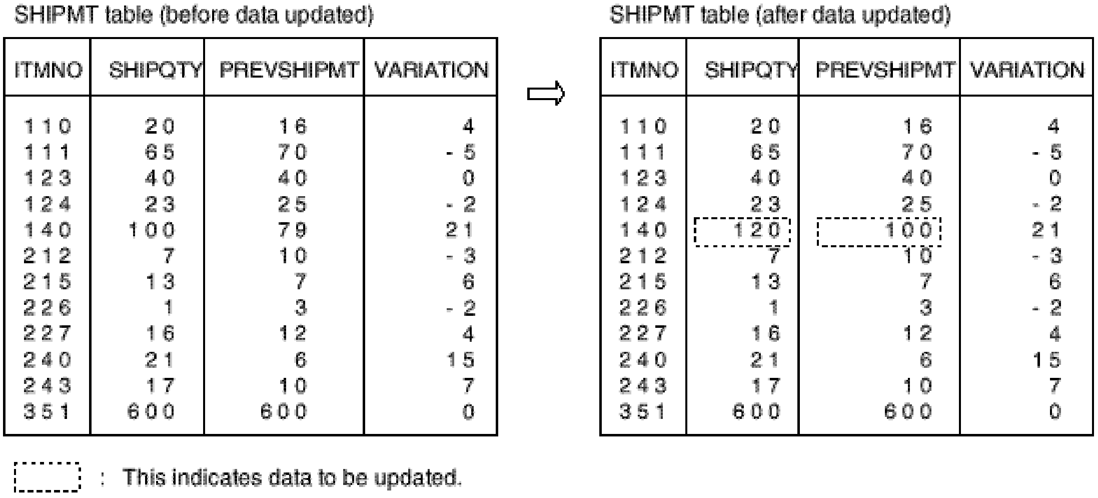
Suppose that a table named SHIPMT table consisting of the four columns ITMNO, SHIPQTY, PREVSHIPMT, and VARIATION exists. When products are shipped, the quantity is set in SHIPQTY and the quantity previously shipped is set in PREVSHIPMT. VARIATION is the difference between the current SHIPQTY and the previous SHIPQTY. The value of VARIATION is not set every time shipment data is produced. Instead, data is assembled and set only when required.

Example:

SHIPQTY is updated to "120" for the product with ITMNO "140" in the SHIPMT table, and the previous SHIPQTY is set for PREVSHIPMT. Column VARIATION is not updated at this time.

```
UPDATE  STOCKS. SHIPMT SET  SHIPQTY = 120,  PREVSHIPMT = SHIPQTY

      WHERE  ITMNO = 140
```

[Figure: Example of using values from other columns to update data]



SHIPMT table (before data updated)

ITMNO	SHIPQTY	PREVSHIPMT	VARIATION
110	20	16	4
111	65	70	-5
123	40	40	0
124	23	25	-2
140	100	79	21
212	7	10	-3
215	13	7	6
226	1	3	-2
227	16	12	4
240	21	6	15
243	17	10	7
351	600	600	0

SHIPMT table (after data updated)

ITMNO	SHIPQTY	PREVSHIPMT	VARIATION
110	20	16	4
111	65	70	-5
123	40	40	0
124	23	25	-2
140	120	100	21
212	7	10	-3
215	13	7	6
226	1	3	-2
227	16	12	4
240	21	6	15
243	17	10	7
351	600	600	0

: This indicates data to be updated.

Note that when column values are specified for data to be updated, the values used are those prior to execution of the UPDATE statement. In this example, SHIPQTY is updated to "120", as specified by the literal. However, PREVSHIPMT is updated to "100", the value of SHIPQTY prior to execution of the UPDATE statement. SHIPQTY is not "120", which is the value to be updated to by this UPDATE statement.

3.2.6 Performing arithmetic operations on data to be updated

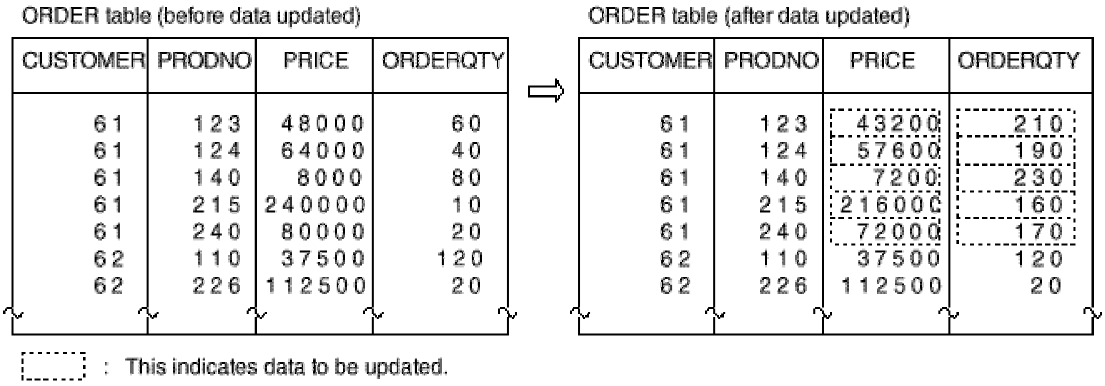
The result of arithmetic operations performed on values such as column values can be used for data to be updated. The following is an example of using this for data to be updated:

Example:

In this example, PRICE is reduced by 10% and 150 is added to ORDERQTY for the data for CUSTOMER "61" in the ORDER table.

```
UPDATE STOCKS. ORDER
       SET PRICE = PRICE * 0.9, ORDERQTY = ORDERQTY + 150
       WHERE CUSTOMER = 61
```

[Figure: Example in which arithmetic operations are performed on data to be updated]



ORDER table (before data updated)

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	123	48000	60
61	124	64000	40
61	140	8000	80
61	215	240000	10
61	240	80000	20
62	110	37500	120
62	226	112500	20

ORDER table (after data updated)

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	123	43200	210
61	124	57600	190
61	140	7200	230
61	215	216000	160
61	240	72000	170
62	110	37500	120
62	226	112500	20

: This indicates data to be updated.

In this example, the operation "PRICE x 0.9" is performed to reduce PRICE by 10%. The operation "PRICE - (PRICE x 0.1)" could also be used. However, the processing efficiency of SQL statements deteriorates as the number of operators increases. As in this example, simplify the operational expression as much as possible.

3.2.7 Updating all rows in a table

To update all rows in a table, do not specify any search conditions or the keyword "WHERE".

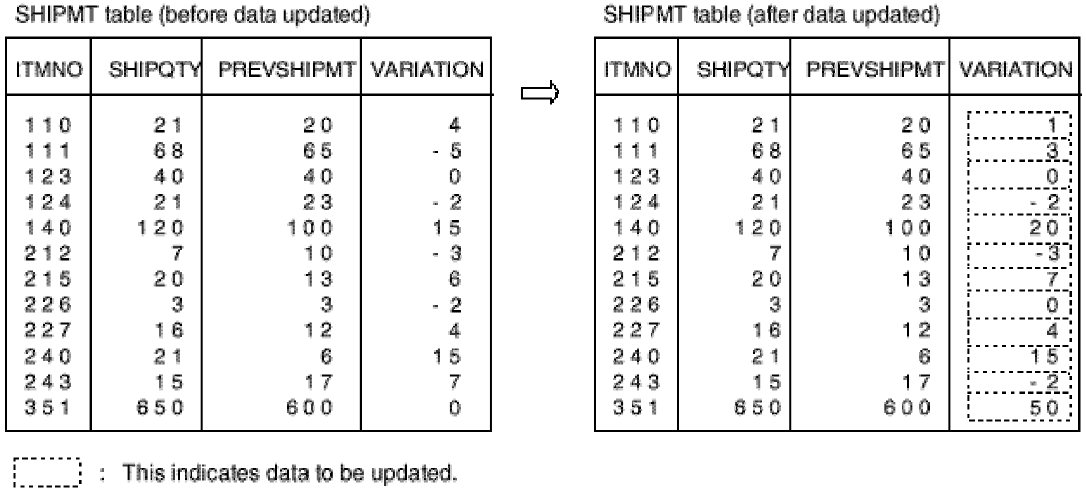
Review the example in Figure: Example of using values from other columns to update data. In this example, when shipment data was produced, the values for SHIPQTY and PREVSHIPMT were updated. However, VARIATION was not specified. In the following example, VARIATION is specified to be updated to the current value. The following is an example of specifying a value for VARIATION:

Example:

In this example, the value of VARIATION in all rows in the SHIPMT table is updated to the value derived by subtracting PREVSHIPMT from SHIPQTY.

```
UPDATE STOCKS. SHIPMT SET VARIATION = SHIPQTY - PREVSHIPMT
                                                   ↑
                                          Do not specify WHERE.
```

[Figure: Example of updating all rows in a table]



SHIPMT table (before data updated)

ITMNO	SHIPQTY	PREVSHIPMT	VARIATION
110	21	20	4
111	68	65	-5
123	40	40	0
124	21	23	-2
140	120	100	15
212	7	10	-3
215	20	13	6
226	3	3	-2
227	16	12	4
240	21	6	15
243	15	17	7
351	650	600	0

⇨

SHIPMT table (after data updated)

ITMNO	SHIPQTY	PREVSHIPMT	VARIATION
110	21	20	1
111	68	65	3
123	40	40	0
124	21	23	-2
140	120	100	20
212	7	10	-3
215	20	13	7
226	3	3	0
227	16	12	4
240	21	6	15
243	15	17	-2
351	650	600	50

┆ ┆ : This indicates data to be updated.

Note: Even in rows in which values do not change, the values are updated to their original values.

In this example, the value of VARIATION was updated in all rows. However, for rows for which shipment data was not produced since the previous update, the actual values did not change. Nevertheless, data in these rows was updated. In other words, the value was updated to a value equal to the original value. The ratio of rows in which the actual value does not change may be large compared to the row count for the entire table. Therefore, using a method to manage rows such that only the required rows are updated, would improve processing efficiency.

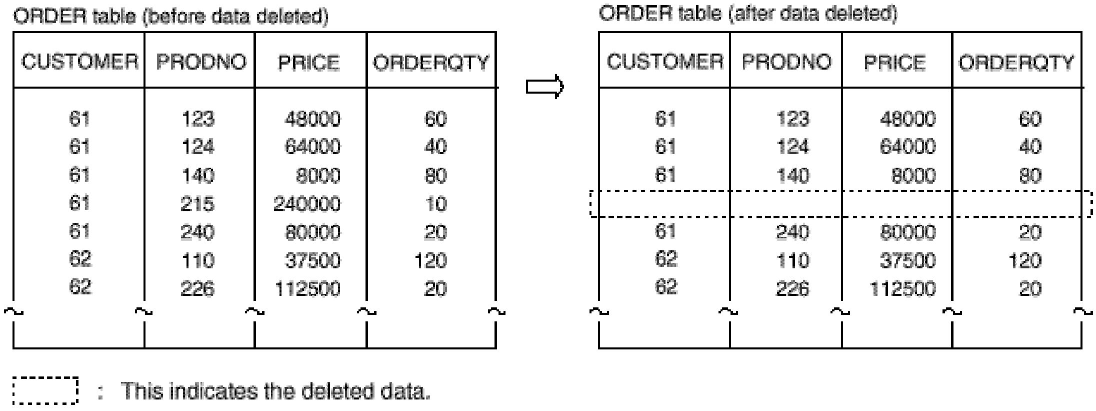
3.3 Deleting Data from a Data Base

To delete data from a table, use the DELETE statement. The following is an example of the DELETE statement:

Example:

In this example, the row for CUSTOMER 61 and PRODNO 215 is deleted from the ORDER table.

```
DELETE FROM STOCKS. ORDER WHERE CUSTOMER = 61 AND PRODNO = 215
            \_____________/       \_____________________________/
              Table name                Search condition
```



ORDER table (before data deleted)

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	123	48000	60
61	124	64000	40
61	140	8000	80
61	215	240000	10
61	240	80000	20
62	110	37500	120
62	226	112500	20

ORDER table (after data deleted)

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	123	48000	60
61	124	64000	40
61	140	8000	80
61	240	80000	20
62	110	37500	120
62	226	112500	20

: This indicates the deleted data.

This section explains the following operation that can be performed by the DELETE statement:

- Deleting all rows from a table

A cursor can also be used to delete data. For information on using a cursor to delete data, see 4.6 "Deleting Data Using a Cursor".

The method of deleting rows explained in this section is called the "DELETE statement" (searched) because the rows to be deleted are specified using search conditions.

3.3.1 Deleting all rows from a table

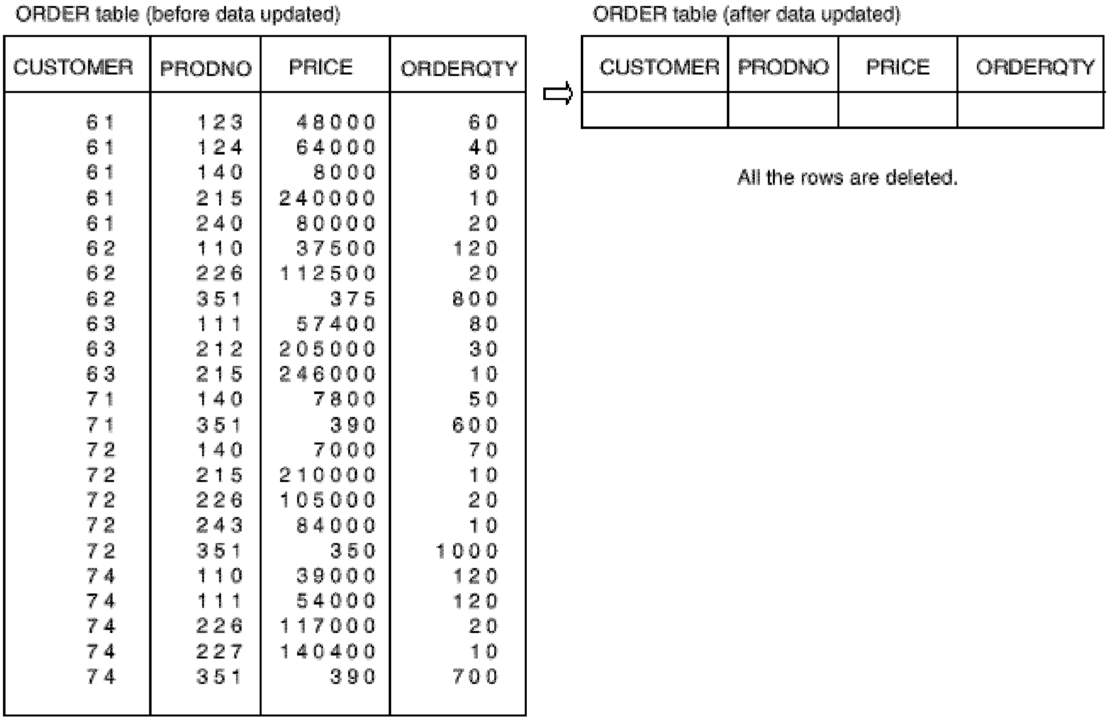
To delete all rows from a table, do not specify search conditions or the keyword "WHERE". The following is an example of this:

Example:

In this example, all rows are deleted from the ORDER table.

```
DELETE  FROM  STOCKS. ORDER
                              ↑
                       Do not specify WHERE.
```

[Figure: Example of deleting all rows from a table]



ORDER table (before data updated)

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	123	48000	60
61	124	64000	40
61	140	8000	80
61	215	240000	10
61	240	80000	20
62	110	37500	120
62	226	112500	20
62	351	375	800
63	111	57400	80
63	212	205000	30
63	215	246000	10
71	140	7800	50
71	351	390	600
72	140	7000	70
72	215	210000	10
72	226	105000	20
72	243	84000	10
72	351	350	1000
74	110	39000	120
74	111	54000	120
74	226	117000	20
74	227	140400	10
74	351	390	700

⇨

ORDER table (after data updated)

CUSTOMER	PRODNO	PRICE	ORDERQTY

All the rows are deleted.

Chapter 4 Using a Cursor to Manipulate Data

This chapter describes how to use a cursor to manipulate data.

This chapter contains the following sections:

4.1 Cursor Overview

4.2 Declaring a Cursor

4.3 Opening and Closing a Cursor

4.4 Positioning a Cursor and Fetching Data

4.5 Updating Data Using a Cursor

4.6 Deleting Data Using a Cursor

4.1 Cursor Overview

A cursor is a virtual tool used to specify one row in a table. A cursor can be used in fetching, updating, or deleting data in a row. Using a cursor to specify a row is called positioning the cursor.

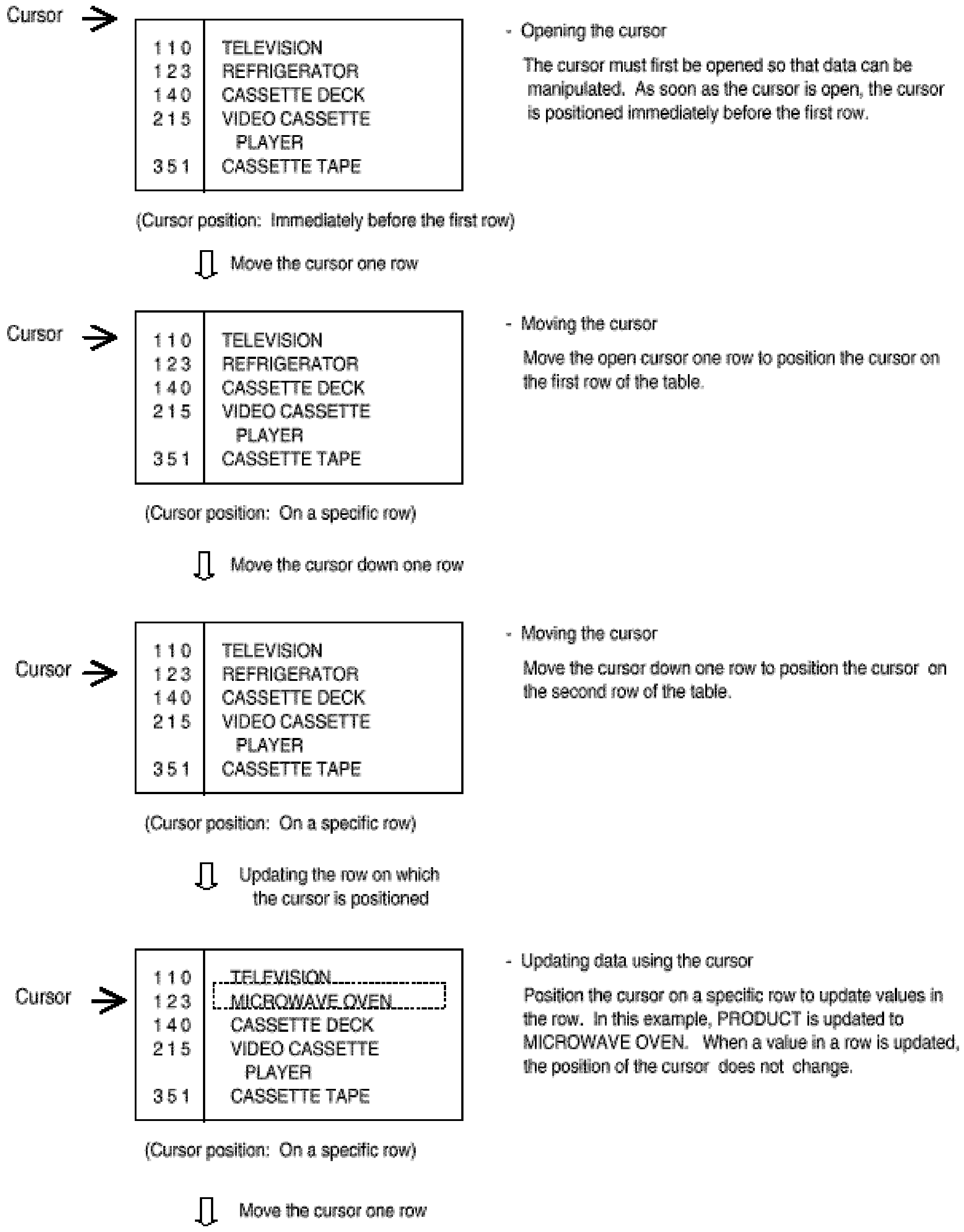
A cursor is either in an open or closed state. If a cursor has never been opened, it is in a state equivalent to a closed state. If a cursor can be used to manipulate data, it is in an open state.

When a cursor is in an open state, it may or may not be positioned on a specific row in a table. Understanding that a cursor may be considered to be in the following four positions helps provide an understanding of the cursor:

- Immediately before the first row
- On a row
- Between two rows
- Immediately after the last row

Figure: Cursor overview provides a simple overview of the cursor using an abbreviated the STOCK table. Column names have not been included in Figure: Cursor overview. The SQL statements used for various data manipulations are explained later in this manual.

[Figure: Cursor overview]



110	TELEVISION
123	REFRIGERATOR
140	CASSETTE DECK
215	VIDEO CASSETTE PLAYER
351	CASSETTE TAPE

(Cursor position: Immediately before the first row)

- Opening the cursor

  The cursor must first be opened so that data can be manipulated. As soon as the cursor is open, the cursor is positioned immediately before the first row.

⇩ Move the cursor one row

110	TELEVISION
123	REFRIGERATOR
140	CASSETTE DECK
215	VIDEO CASSETTE PLAYER
351	CASSETTE TAPE

(Cursor position: On a specific row)

- Moving the cursor

  Move the open cursor one row to position the cursor on the first row of the table.

⇩ Move the cursor down one row

110	TELEVISION
123	REFRIGERATOR
140	CASSETTE DECK
215	VIDEO CASSETTE PLAYER
351	CASSETTE TAPE

(Cursor position: On a specific row)

- Moving the cursor

  Move the cursor down one row to position the cursor on the second row of the table.

⇩ Updating the row on which the cursor is positioned

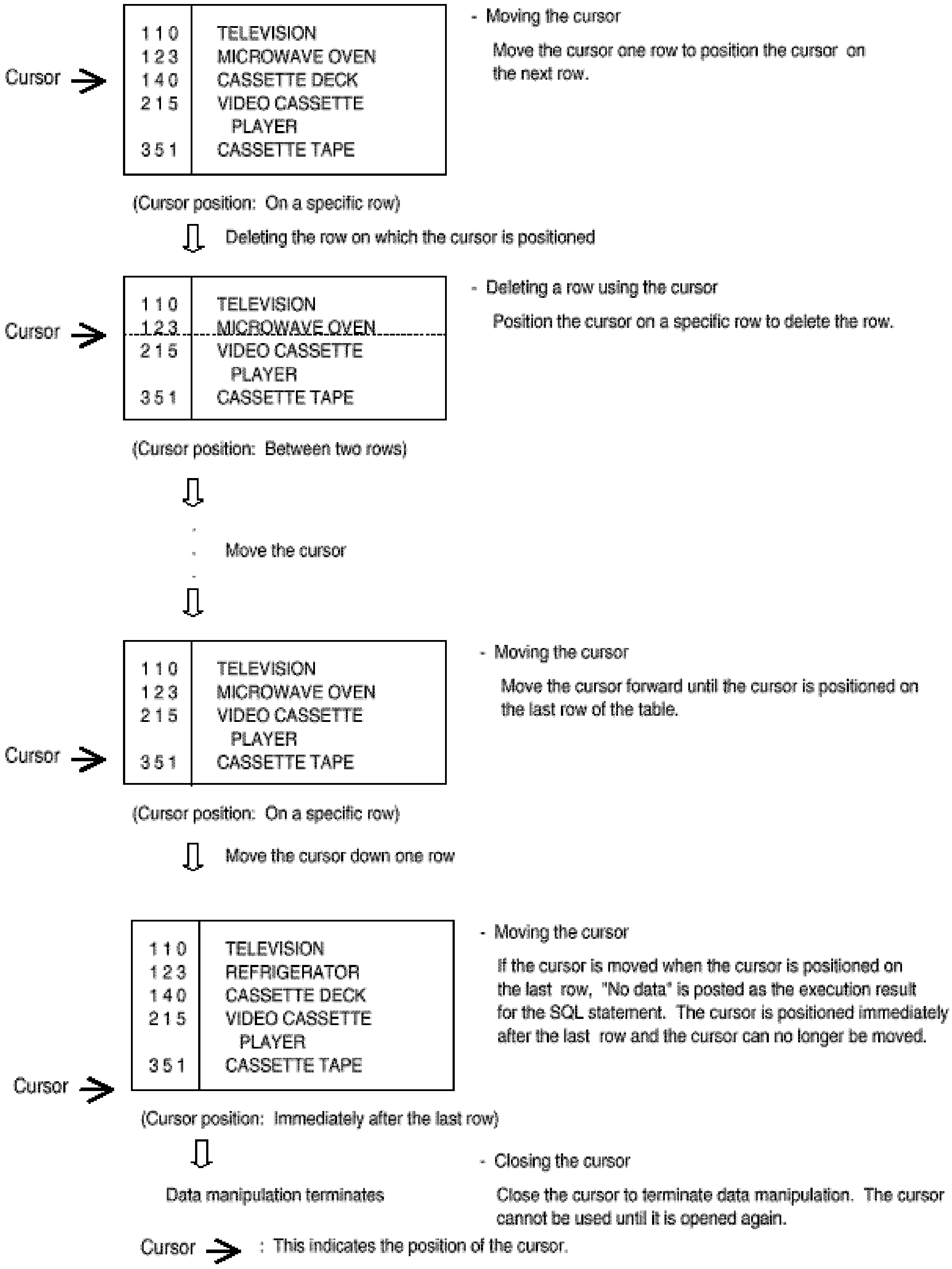
110	TELEVISION
123	MICROWAVE OVEN
140	CASSETTE DECK
215	VIDEO CASSETTE PLAYER
351	CASSETTE TAPE

(Cursor position: On a specific row)

- Updating data using the cursor

  Position the cursor on a specific row to update values in the row. In this example, PRODUCT is updated to MICROWAVE OVEN. When a value in a row is updated, the position of the cursor does not change.

⇩ Move the cursor one row


Cursor
110 TELEVISION
123 MICROWAVE OVEN
140 CASSETTE DECK
215 VIDEO CASSETTE PLAYER
351 CASSETTE TAPE
Cursor
110 TELEVISION
123 MICROWAVE OVEN
215 VIDEO CASSETTE PLAYER
351 CASSETTE TAPE
Cursor
110 TELEVISION
123 MICROWAVE OVEN
215 VIDEO CASSETTE PLAYER
351 CASSETTE TAPE
Cursor
110 TELEVISION
123 REFRIGERATOR
140 CASSETTE DECK
215 VIDEO CASSETTE PLAYER
351 CASSETTE TAPE
Cursor

- Moving the cursor

Move the cursor one row to position the cursor on the next row.

(Cursor position: On a specific row)

Deleting the row on which the cursor is positioned

- Deleting a row using the cursor

Position the cursor on a specific row to delete the row.

(Cursor position: Between two rows)

Move the cursor

- Moving the cursor

Move the cursor forward until the cursor is positioned on the last row of the table.

(Cursor position: On a specific row)

Move the cursor down one row

- Moving the cursor

If the cursor is moved when the cursor is positioned on the last row, "No data" is posted as the execution result for the SQL statement. The cursor is positioned immediately after the last row and the cursor can no longer be moved.

(Cursor position: Immediately after the last row)

Data manipulation terminates

- Closing the cursor

Close the cursor to terminate data manipulation. The cursor cannot be used until it is opened again.

Cursor → : This indicates the position of the cursor.

■Sequence of data manipulations using a cursor

Cursor SQL statements are as follows:

- Cursor declaration (DECLARE CURSOR)
- OPEN statement
- CLOSE statement
- FETCH statement
- UPDATE statement
- DELETE statement

The SQL statement used to declare a cursor is the cursor declaration, a non-executable statement. The cursor declaration specifies the tables in which data is to be manipulated and the method by which data is to be fetched. Before data can be manipulated using a cursor, the cursor must be declared. The cursor declaration must be coded before any SQL statements that use the cursor are coded, regardless of the execution sequence of the application program.

Data is manipulated in the following sequence using a cursor:

1. The OPEN statement opens the cursor.
2. The FETCH statement positions the cursor at a specific row in the table and fetches data.
3. The data in the row where the cursor is positioned is processed.
    - Use an application program to process fetched data.
    - Use the UPDATE statement to update values in the row where the cursor is positioned.
    - Use the DELETE statement to delete the row where the cursor is positioned.
4. The CLOSE statement closes the cursor.

To process more than one row, repeat steps 2) and 3). These steps are illustrated in the flowchart in Figure: Flowchart of using cursor to manipulation data.

[Figure: Flowchart of using cursor to manipulation data]

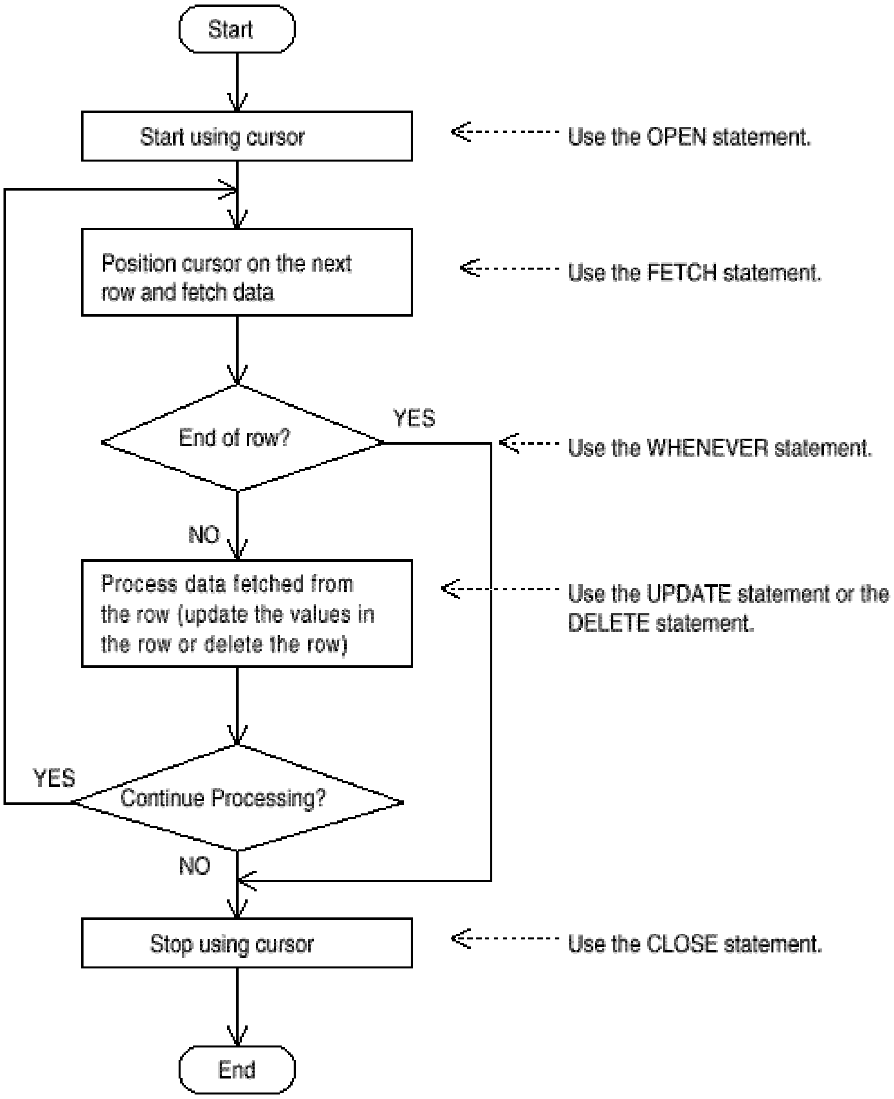
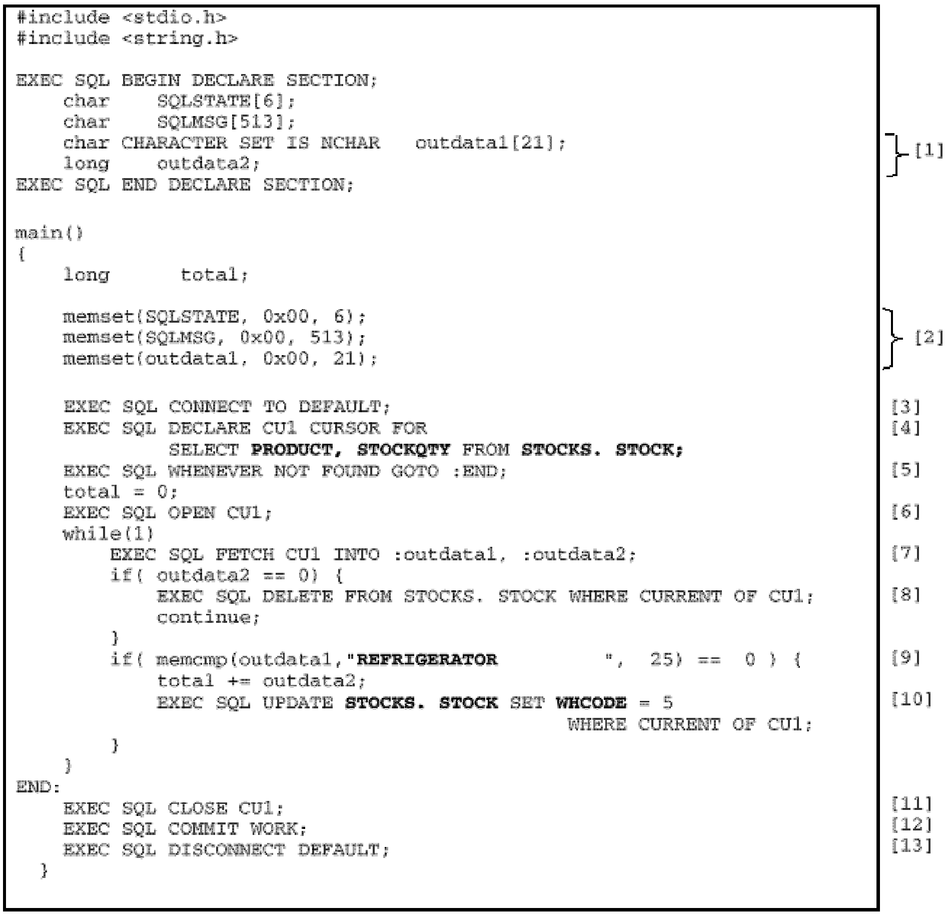

Start
Start using cursor
Use the OPEN statement.
Position cursor on the next row and fetch data
Use the FETCH statement.
End of row?
YES
Use the WHENEVER statement.
NO
Process data fetched from the row (update the values in the row or delete the row)
Use the UPDATE statement or the DELETE statement.
YES
Continue Processing?
NO
Stop using cursor
Use the CLOSE statement.
End

Figure: Example of an application program that uses a cursor to manipulate data is an example of an application program that uses a cursor to manipulate data. The application program processes the STOCK table as follows:

- Deletes rows in which STOCKQTY is "0"
- Changes WHCODE to "5" in rows for which PRODUCT is REFRIGERATOR.

- Calculates the total of STOCKQTY in the rows for which PRODUCT is REFRIGERATOR and stores the value of total in variable TOTAL.

[Figure: Example of an application program that uses a cursor to manipulate data]



```
#include <stdio.h>
#include <string.h>

EXEC SQL BEGIN DECLARE SECTION;
    char    SQLSTATE[6];
    char    SQLMSG[513];
    char CHARACTER SET IS NCHAR   outdata1[21];                              } [1]
    long    outdata2;
EXEC SQL END DECLARE SECTION;

main()
{
    long      total;

    memset(SQLSTATE, 0x00, 6);
    memset(SQLMSG, 0x00, 513);                                               } [2]
    memset(outdata1, 0x00, 21);

    EXEC SQL CONNECT TO DEFAULT;                                               [3]
    EXEC SQL DECLARE CU1 CURSOR FOR                                            [4]
             SELECT PRODUCT, STOCKQTY FROM STOCKS. STOCK;
    EXEC SQL WHENEVER NOT FOUND GOTO :END;                                     [5]
    total = 0;
    EXEC SQL OPEN CU1;                                                         [6]
    while(1)
        EXEC SQL FETCH CU1 INTO :outdata1, :outdata2;                          [7]
        if( outdata2 == 0) {
            EXEC SQL DELETE FROM STOCKS. STOCK WHERE CURRENT OF CU1;           [8]
            continue;
        }
        if( memcmp(outdata1,"REFRIGERATOR         ",  25) ==  0 ) {            [9]
            total += outdata2;
            EXEC SQL UPDATE STOCKS. STOCK SET WHCODE = 5                       [10]
                                               WHERE CURRENT OF CU1;
        }
    }
END:
    EXEC SQL CLOSE CU1;                                                        [11]
    EXEC SQL COMMIT WORK;                                                      [12]
    EXEC SQL DISCONNECT DEFAULT;                                               [13]
  }
```

[1] The host variables in which resulting data is to be stored are declared.

[2] The host variables are initialized.

[3] The connection is made.

[4] The declaration statement for cursor CU1 is specified. Cursor CU1 is declared for a table consisting of columns PRODUCT and STOCKQTY from the STOCK table.

[5] When the cursor is positioned as at step [7] and the cursor has advanced to the end of the data, the processing should jump to the label END.

[6] Cursor CU1 starts to be used. The cursor table declared at step [4] is created, and the cursor is positioned immediately before the first row of that table.

[7] Cursor CU1 is positioned on the next row or on the first row of the table if the statement is executed immediately after the OPEN statement. Data is fetched from the row on which the cursor is positioned. Data for PRODUCT is stored in outdata1 and data for STOCKQTY is stored in outdata2.

[8] The execution results for the statement in step [7] are checked. If STOCKQTY is 0, the row is deleted.

[9] The execution results for the statement in step [7] are checked. If PRODUCT is REFRIGERATOR, the value of STOCKQTY (stored in outdata2) is added to TOTAL.

[10] WHCODE is changed to 5 in the row where the cursor is positioned.

[11] Use of the cursor ends.

[12] The current transaction terminates.

[13] The connection is broken.

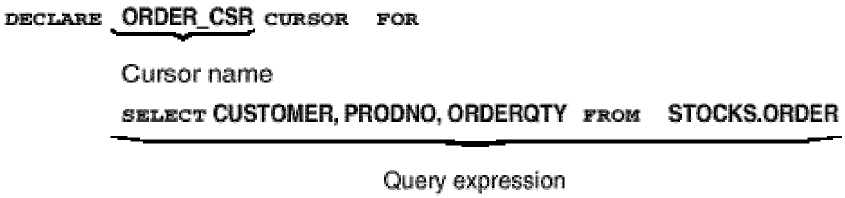
4.2 Declaring a Cursor

Before data can be manipulated using a cursor, the cursor must be declared using a cursor declaration, a non-executable statement. The cursor declaration must be coded before any SQL statements that use the cursor are coded, regardless of the execution sequence of the application program.

The following is an example of a cursor declaration:

Example 1:

In this example, a cursor to obtain CUSTOMER, PRODNO, and ORDERQTY from the ORDER table is declared.



```
DECLARE ORDER_CSR CURSOR FOR
        SELECT CUSTOMER, PRODNO, ORDERQTY FROM STOCKS.ORDER
```

This cursor declaration declares a cursor to fetch values from all rows of the three columns CUSTOMER, PRODNO, and ORDERQTY in the ORDER table.

The tables to be manipulated by a cursor and the method of fetching data are specified in the query expression of the cursor declaration. Example 1 is a simple example of data manipulation performed by a query expression. In contrast, a cursor can be used to fetch data from tables and perform various processes according to the application program logic. A query expression can be used to group and manipulate tables and fetch data from the aggregate of multiple tables. The skillful use of query expressions can greatly reduce the amount of processing required to be performed by application programs.

■Query expression

Query expressions can be specified in two ways. A single query specification can be specified in a query expression. Alternatively, two or more query specifications, joined by UNION, can be specified in a query expression. For information on how to specify "UNION", see 5.5 "Obtaining the Aggregate for Rows from Multiple Tables".

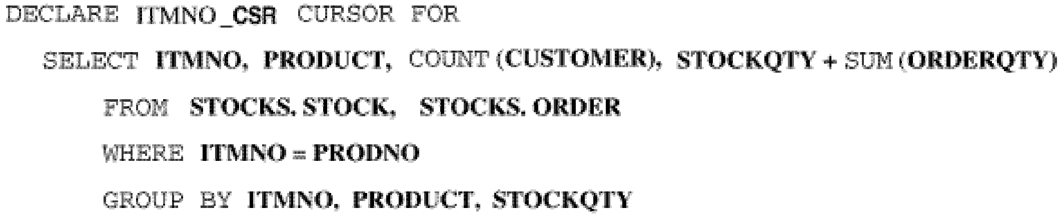
■Query specification

The format of a query specification is the same as that for a single row SELECT statement minus the INTO clause. A query specification derives a table in the same way as a single row SELECT statement. In both cases, values for columns specified in a select column list are fetched from rows of a table, derived from a FROM clause, that satisfy the WHERE clause conditions. However, whereas a single row SELECT statement ends in an error if more than one row is to be fetched, a query specification can be used to fetch any number of rows.

The GROUP BY clause and the HAVING clause can be specified in query specifications. For details, see 6.1 "Grouping Tables and Manipulating Data", which explains using a cursor declaration containing a GROUP BY clause in the query specification. The following is an example of specifying a GROUP BY clause:

Example 2:

In this example, a cursor is declared to obtain PRODUCT, the number of customers who have placed orders, and the totals of STOCKQTY and ORDERQTY for each ITMNO in the STOCK table and the ORDER table.



```
DECLARE ITMNO_CSR CURSOR FOR
   SELECT ITMNO, PRODUCT, COUNT (CUSTOMER), STOCKQTY + SUM (ORDERQTY)
       FROM STOCKS. STOCK, STOCKS. ORDER
       WHERE ITMNO = PRODNO
       GROUP BY ITMNO, PRODUCT, STOCKQTY
```

[Figure: Deriving a cursor table]

STOCK table

ITMNO	PRODUCT	STOCKQTY	WHCODE
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
140	CASSETTE DECK	120	2
212	TELEVISION	0	2
215	VIDEO CASSETTE PLAYER	5	2
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETTE TAPE	2500	2

ORDER table

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	123	48000	60
61	124	64000	40
61	140	8000	80
61	215	240000	10
61	240	80000	20
62	110	37500	120
62	226	112500	20
62	351	375	800
63	111	57400	80
63	212	205000	30
63	215	246000	10
71	140	7800	50
71	351	390	600
72	140	7000	70
72	215	210000	10
72	226	105000	20
72	243	84000	10
72	351	350	1000
74	110	39000	120
74	111	54000	120
74	226	117000	20
74	227	140400	10
74	351	390	700

⇩

Cursor table

ITMNO	PRODUCT	(*1)	(*2)
110	TELEVISION	2	325
111	TELEVISION	2	290
123	REFRIGERATOR	1	120
124	REFRIGERATOR	1	115
140	CASSETTE DECK	3	320
212	TELEVISION	1	30
215	VIDEO CASSETTE PLAYER	3	35
226	REFRIGERATOR	3	68
227	REFRIGERATOR	1	25
240	CASSETTE DECK	1	45
243	CASSETTE DECK	1	24
351	CASSETTE TAPE	4	5600

*1 This column is created from the result of COUNT (CUSTOMER).

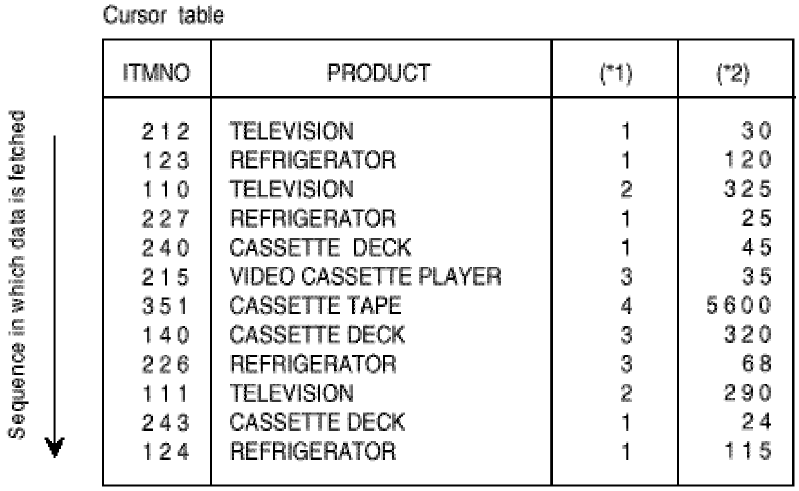
*2 This column is created from the result of STOCKQTY + SUM (ORDERQTY).

Note: Columns containing the results of operational expressions and set functions do not have names.

■Sequence of rows fetched using a cursor

Note that the sequence of rows fetched using a cursor was not specified. Although the query expression specifies the rows to be included in the cursor table, the query expression does not specify the sequence of fetching the rows. This is true even for a simple cursor that fetches all rows of a table. The rows in the cursor table shown in Figure: Deriving a cursor table are sometimes arranged in ITMNO sequence. However, when an application program is actually executed, the sequence of rows may be as shown in Figure: Example of rows in a cursor table when the sequence is undefined.

[Figure: Example of rows in a cursor table when the sequence is undefined]



Cursor table

Sequence in which data is fetched

ITMNO	PRODUCT	(*1)	(*2)
212	TELEVISION	1	30
123	REFRIGERATOR	1	120
110	TELEVISION	2	325
227	REFRIGERATOR	1	25
240	CASSETTE DECK	1	45
215	VIDEO CASSETTE PLAYER	3	35
351	CASSETTE TAPE	4	5600
140	CASSETTE DECK	3	320
226	REFRIGERATOR	3	68
111	TELEVISION	2	290
243	CASSETTE DECK	1	24
124	REFRIGERATOR	1	115

*1 This column is created from the result of COUNT (CUSTOMER).

*2 This column is created from the result of STOCKQTY + SUM (ORDERQTY).

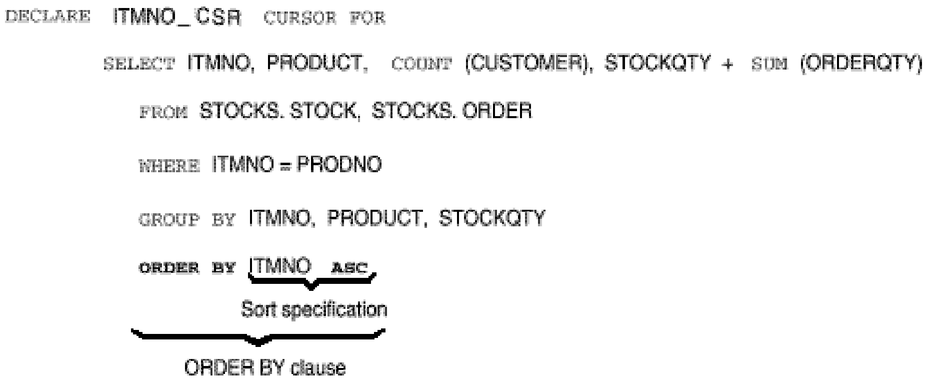
Note: Columns containing the results of operational expressions and set functions do not have names.

■Reordering the sequence of rows

A cursor declaration can specify the reordering, or sorting, of rows in a cursor table. This is done by specifying the ORDER BY clause after the query expression. The following is an example of specifying the ORDER BY clause:

Example 3:

In this example, the cursor declaration used in Example 2 specifies that data is fetched by the cursor in the sequence from lowest ITMNO.



```
DECLARE ITMNO_CSR CURSOR FOR
        SELECT ITMNO, PRODUCT, COUNT (CUSTOMER), STOCKQTY + SUM (ORDERQTY)
          FROM STOCKS. STOCK, STOCKS. ORDER
          WHERE ITMNO = PRODNO
          GROUP BY ITMNO, PRODUCT, STOCKQTY
          ORDER BY ITMNO ASC
```

Sort specification
ORDER BY clause

Specify the names of the columns used for sorting and the method of sorting in the ORDER BY clause. The columns used for sorting must be columns contained in the sort table. In other words, only columns derived from the result of the query expression can be used in sorting. Columns used in sorting are called sort keys. Sorting can be performed in either ascending (ASC) or descending (DESC) order. If no sort method is specified, the system uses "ASC" as the default.

ASC:

This reorders values starting with the smallest value. This is called sorting in ascending order.

DESC:

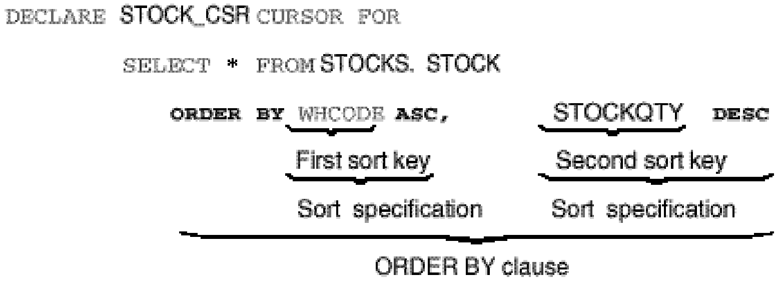
This reorders values starting with the largest value. This is called sorting in descending order.

Figure: Deriving a cursor table shows the sequence of rows in the cursor table if the cursor declaration is specified as in Example 3. In Figure: Deriving a cursor table, the value of ITMNO is different for each row, so the sequence of rows in the cursor table is unique. The situation when the value of ITMNO is the same for more than one row is explained further on.

The sort specification determines the sequence of rows that have different values for the sort key. However, if more than one row has the same value for the sort key, the sequence of rows cannot be determined. To reorder rows that have the same value for the sort key, specify multiple sort keys separated by commas. Multiple sort keys specified in the ORDER BY clause are called the first sort key, the second sort key, and so on. Rows in the cursor table are first reordered by the first sort key, then rows that have the same value for the first sort key are reordered by the second sort key. The sequence of rows that have the same values for all specified sort keys cannot be determined. The following is an example of specifying sort keys:

Example 4:

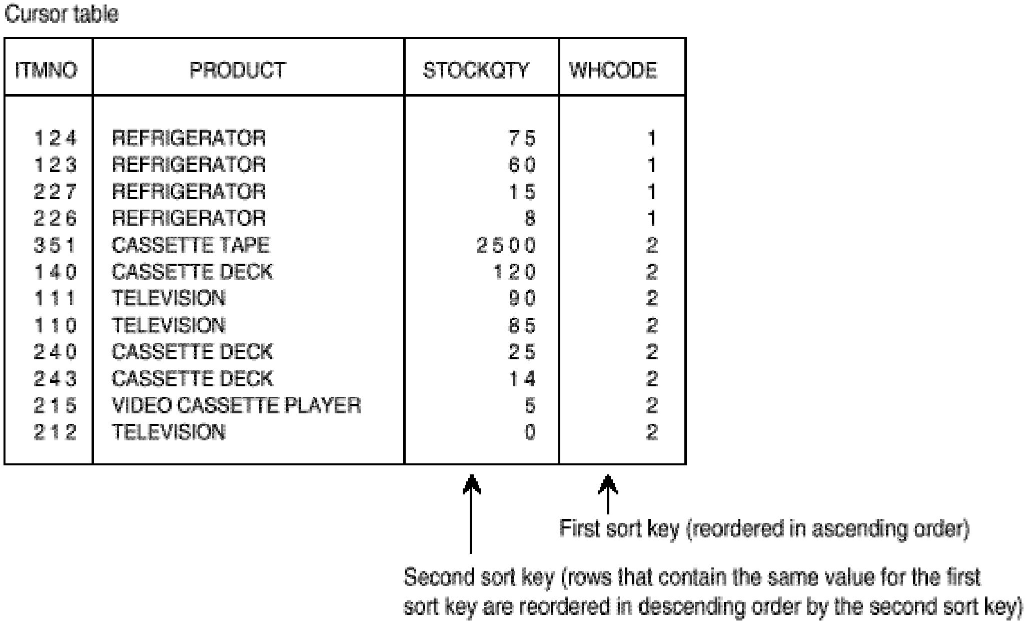
> In this example, the cursor declaration specifies that data is fetched by cursor in the sequence from lowest WHCODE.



```
DECLARE STOCK_CSR CURSOR FOR
        SELECT * FROM STOCKS. STOCK
             ORDER BY WHCODE ASC,      STOCKQTY  DESC
```

First sort key
Second sort key
Sort specification
Sort specification
ORDER BY clause

[Figure: Example of specifying multiple sort keys]



Cursor table

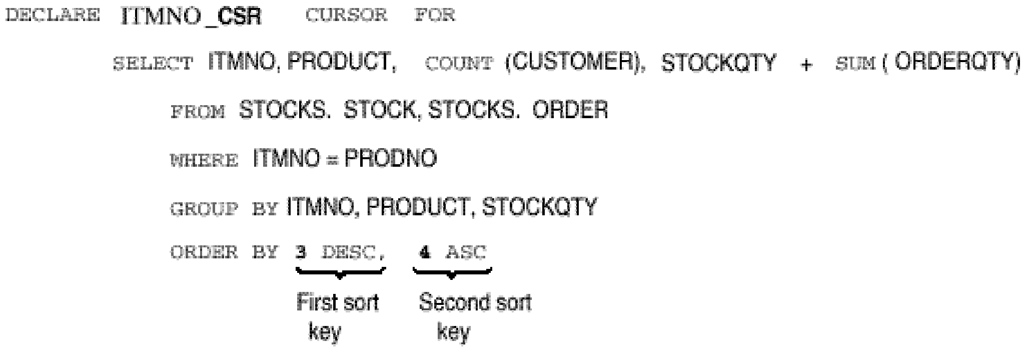
ITMNO	PRODUCT	STOCKQTY	WHCODE
124	REFRIGERATOR	75	1
123	REFRIGERATOR	60	1
227	REFRIGERATOR	15	1
226	REFRIGERATOR	8	1
351	CASSETTE TAPE	2500	2
140	CASSETTE DECK	120	2
111	TELEVISION	90	2
110	TELEVISION	85	2
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
215	VIDEO CASSETTE PLAYER	5	2
212	TELEVISION	0	2

First sort key (reordered in ascending order)
Second sort key (rows that contain the same value for the first sort key are reordered in descending order by the second sort key)

When rows are to be reordered, the names of columns used as sort keys must be specified. However, columns containing the results of operational expressions or set functions specified in a select column list in a query specification do not have names. To use a column without a name as a sort key, specify a column number instead of a column name in the sort key specification. Columns are numbered in sequence from the left in the cursor table. The following is an example of specifying column numbers:

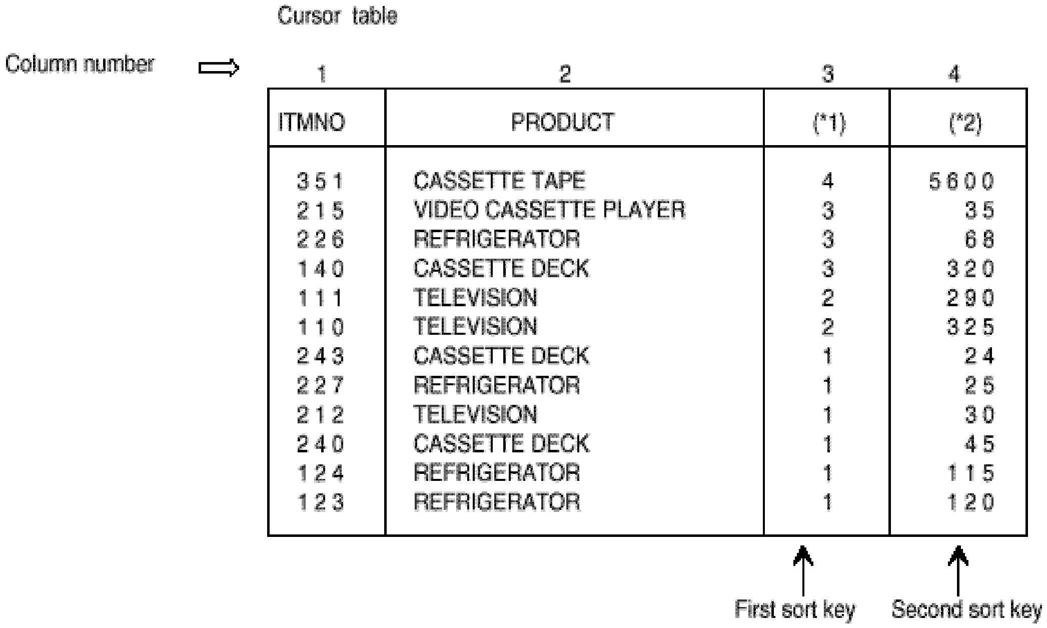
Example 5:

In this example, the cursor declaration used in Example 2 is used. Here, data is fetched by the cursor using the number of CUSTOMERs as the first sort key. The data is sorted in descending order. The totals of STOCKQTY and ORDERQTY are used as the second sort key, and the data is sorted in ascending order.



```
DECLARE ITMNO_CSR   CURSOR  FOR
          SELECT ITMNO, PRODUCT,  COUNT (CUSTOMER), STOCKQTY  +  SUM ( ORDERQTY)
              FROM STOCKS. STOCK, STOCKS. ORDER
              WHERE ITMNO = PRODNO
              GROUP BY ITMNO, PRODUCT, STOCKQTY
              ORDER BY 3 DESC,  4 ASC
```

[Figure: Example of specifying column numbers in sort specification]



Cursor table
Column number
First sort key
Second sort key

1	2	3	4
ITMNO	PRODUCT	(*1)	(*2)
351	CASSETTE TAPE	4	5600
215	VIDEO CASSETTE PLAYER	3	35
226	REFRIGERATOR	3	68
140	CASSETTE DECK	3	320
111	TELEVISION	2	290
110	TELEVISION	2	325
243	CASSETTE DECK	1	24
227	REFRIGERATOR	1	25
212	TELEVISION	1	30
240	CASSETTE DECK	1	45
124	REFRIGERATOR	1	115
123	REFRIGERATOR	1	120

*1 This column is created from the result of COUNT (CUSTOMER).

*2 This column is created from the result of STOCKQTY + SUM (ORDERQTY).

Note: Columns containing the results of operational expressions and set functions do not have names.

■Updatable cursors and read-only cursors

Two types of cursors are available: updatable cursor and read-only cursors. Only updatable cursors can be used in UPDATE statements and DELETE statements. Both updatable cursors and read-only cursors can be used in FETCH statements. The cursor declaration determines if a cursor is updatable or read-only.

The following shows cursor declaration conditions for read-only cursors. SCM is used as the schema name for tables TBL1 and TBL2 in this example.

Conditions of cursor declaration for read-only cursors	**Example**
a) DISTINCT is specified.	DECLARE CSR CURSOR FOR SELECT **DISTINCT** COLA, COLB FROM SCM.TBL1
b) An operational expression is specified in the select column list.	DECLARE CSR CURSOR FOR SELECT COLC, **COLA + COLB** FROM SCM.TBL1
c) A set function is specified in the select column list.	DECLARE CSR CURSOR FOR SELECT **SUM(COLB)** FROM SCM.TBL1
d) The same column is specified in the select column list.	DECLARE CSR CURSOR FOR SELECT **COLA**, COLB, **COLA**, COLC FROM SCM.TBL1
e) A literal or host variable is specified in the select column list.	DECLARE CSR CURSOR FOR SELECT COLA, **100** FROM SCM.TBL1
f) A data column function is specified in the select column list.	DECLARE CSR CURSOR FOR SELECT **SUBSTRING(COL1 FROM 1 FOR 3)** FROM SCM.TBL1
g) A numeric function is specified in the select column list.	DECLARE CSR CURSOR FOR SELECT **ABS(COL1)** FROM SCM.TBL1
h) A date-time value function is specified in the select column list.	DECLARE CSR CURSOR FOR SELECT **TRUNC_DATE(COL1,'MONTH')** FROM SCM.TBL1
I) The CASE expression is specified in the select column list.	DECLARE CSR CURSOR FOR SELECT **CASE WHEN COL1=1 THEN 'ONE' END** FROM SCM.TBL1
j) The CAST specification is specified in the select column list.	DECLARE CSR CURSOR FOR SELECT **CAST(COL1 AS CHAR(6))** FROM SCM.TBL1
k) A concatenation operator is specified in the select column list.	DECLARE CSR CURSOR FOR SELECT **COL1 || COL2** FROM SCM.TBL1
l) A function routine is specified in the select column list.	DECLARE CSR CURSOR FOR SELECT **USERFUNC1(C1,C2)** FROM SCM.TBL1
m) A sequence is specified in the select column list.	DECLARE CSR CURSOR FOR SELECT **SCM.SEQUENCE1.NEXTVAL** FROM SCM.TBL1
n) Two or more tables or concatenation tables are specified in the FROM clause.	DECLARE CSR CURSOR FOR SELECT COLA, COLZ FROM **SCM.TBL1, SCM.TBL2**
o) The GROUP BY or HAVING clause is specified.	DECLARE CSR CURSOR FOR SELECT COLA FROM SCM.TBL1 **GROUP BY COLA**
p) A subquery is specified in the WHERE clause.	DECLARE CSR CURSOR FOR SELECT * FROM SCM.TBL1 **WHERE COLA IN** **(SELECT COLZ FROM SCM.TBL2 WHERE COLY > 2000)**
q) UNION is specified.	DECLARE CSR CURSOR FOR SELECT COLC FROM SCM.TBL1 **UNION** SELECT COLX FROM SCM.TBL2
r) FOR READ ONLY is specified in an updatable clause.	DECLARE CSR CURSOR FOR SELECT COLA, COLB FROM SCM.TBL1 **FOR READ ONLY**
s) The ORDER BY clause is specified, and an updatable clause is not specified.	DECLARE CSR CURSOR FOR SELECT COLA, COLB FROM SCM.TBL1 **ORDER BY COLA**
t) The SCROLL clause is specified, and an updatable clause is not specified.	DECLARE CSR **SCROLL** CURSOR FOR SELECT COLA, COLB FROM SCM.TBL1

If one of conditions a) to t) applies to the format of the cursor declaration, the cursor is read-only. If none of the conditions applies, the cursor is updatable.

Conditions a) to p) are specified in query specifications. Condition q) is specified in query expressions. Conditions r) to t) are specified in cursor declarations. Conditions r) and s) support updatable cursors only if FOR UPDATE is specified in the updatable clause. For more information on the subquery in condition p), see 6.2 "Specifying Various Search Conditions". For more information on UNION in condition q), see 5.5 "Obtaining the Aggregate for Rows from Multiple Tables".

4.3 Opening and Closing a Cursor

Before data can be manipulated using a cursor, the cursor must be opened using an OPEN statement. After data has been manipulated, the cursor must be closed using a CLOSE statement.

■Opening a cursor

Use an OPEN statement to begin using a cursor. The cursor table specified in the cursor declaration is created as soon as the OPEN statement is executed. The following is an example in which STOCK_CSR is opened:

$$\texttt{OPEN}\ \ \underbrace{\texttt{STOCK\_CSR}}_{\text{Cursor name}}$$

The cursor declaration specifying the cursor name must be coded before the OPEN statement in the coding sequence for the application program. A cursor specified in an OPEN statement must be one that has never been opened or one that has been closed by a CLOSE statement. If a cursor is opened more than once, an error occurs. Different cursors can be opened at the same time.

■Closing a cursor

Use the CLOSE statement to stop using a cursor. The following is an example of closing STOCK_CSR:

$$\texttt{CLOSE}\ \ \underbrace{\texttt{STOCK\_CSR}}_{\text{Cursor name}}$$

The cursor declaration specifying the cursor name must be coded before the CLOSE statement in the coding sequence for the application program. A cursor specified in a CLOSE statement must be open. If a cursor that is not open is specified in a CLOSE statement, an error occurs.

4.4 Positioning a Cursor and Fetching Data

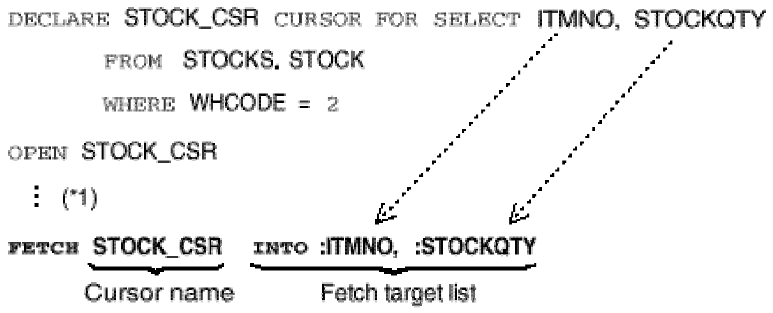
Use the FETCH statement to manipulate data in the following ways:

- To position a cursor and fetch data from the row on which the cursor is positioned
- To position a cursor on the row to be updated and update data in the row using the UPDATE statement
- To position a cursor on a row and delete the row using the DELETE statement

The following is an example of specifying the FETCH statement:

Example 1:

> In this example, a cursor is declared to fetch ITMNO and STOCKQTY for products with WHCODE "2" from the STOCK table. The cursor moves one row at a time to fetch values.



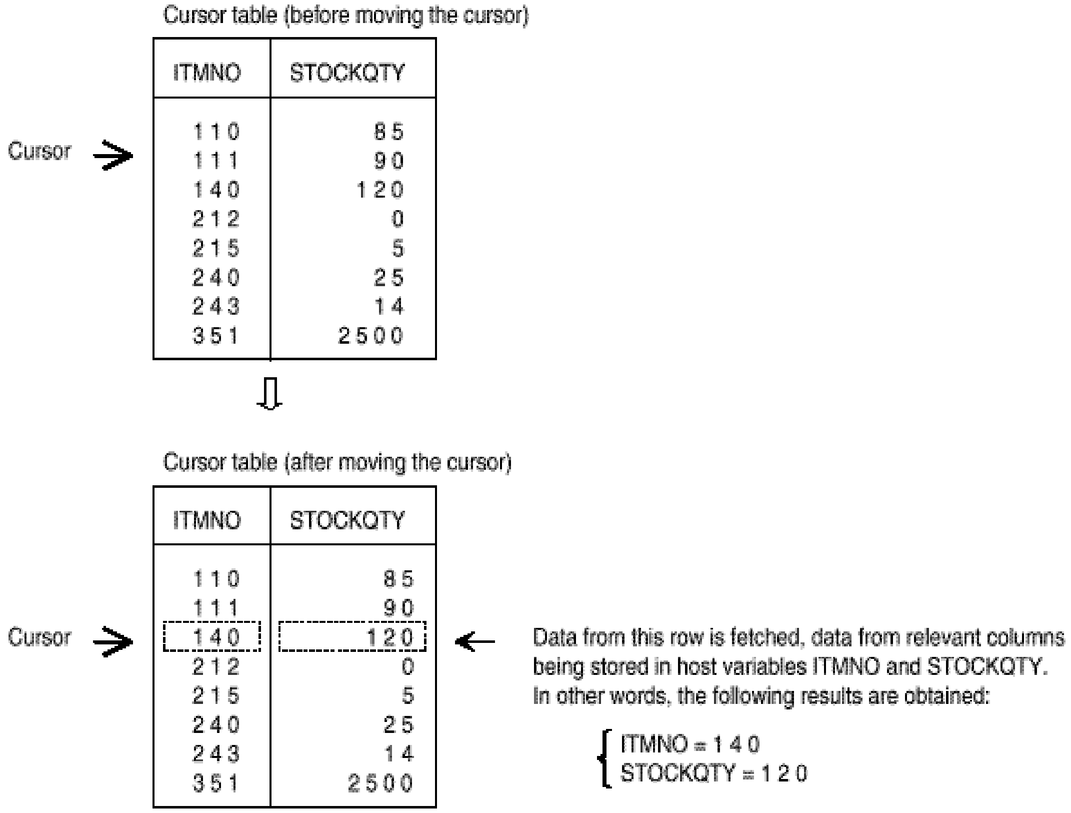
```
DECLARE STOCK_CSR CURSOR FOR SELECT ITMNO, STOCKQTY
        FROM STOCKS. STOCK
        WHERE WHCODE = 2
OPEN STOCK_CSR
  ⋮ (*1)
FETCH STOCK_CSR INTO :ITMNO, :STOCKQTY
```

------> : This indicates the correspondence between the columns from which data is fetched and the host variables in which the fetched data is stored.

*1 The FETCH statement can be repeatedly executed to fetch data from multiple rows of a table. Generally, a statement to control repetition is specified in the host language in this location.

Figure: Example of using a FETCH statement to position a cursor shows the results of executing the FETCH statement in Example 1. The cursor is positioned on the second row because the FETCH statement was executed twice.

[Figure: Example of using a FETCH statement to position a cursor]



Cursor table (before moving the cursor)

ITMNO	STOCKQTY
110	85
111	90
140	120
212	0
215	5
240	25
243	14
351	2500

Cursor table (after moving the cursor)

ITMNO	STOCKQTY
110	85
111	90
140	120
212	0
215	5
240	25
243	14
351	2500

Data from this row is fetched, data from relevant columns being stored in host variables ITMNO and STOCKQTY. In other words, the following results are obtained:

ITMNO = 140
STOCKQTY = 120

Cursor → : This indicates the position of the cursor.

[ ] : This indicates the fetched data.

The cursor declaration specifying the cursor name must be coded before the FETCH statement in the coding sequence for the application program. The cursor specified by the FETCH statement must be open. If a cursor that is not open is specified, an error occurs. If the FETCH statement is executed when the cursor is positioned in the last row, the result is "No data". Likewise, if the FETCH statement is executed when the cursor is positioned immediately after the last row, the result is still "No data".

■Host variable in which fetched data is stored

The host variables in which values are stored for the row in which the cursor is positioned are specified in the fetch target list. The number of host variables specified must be the same as the number of columns in the cursor table. Data from the fetched columns is stored in the host variables in the same sequence from the left.

■Fetching data containing null values

To fetch data containing null values, specify an indicator variable to be paired with the host variable in which fetched data is stored. The following is an example of specifying a FETCH statement to fetch data containing null values:

Example 2:

In this example, indicator variables are specified in the FETCH statement used in Example 1.

```
FETCH  STOCK_CSR INTO  :ITMNO INDICATOR :ITMNO_INDICATOR


                     :STOCKQTY INDICATOR :STOCKQTY_INDICATOR
```

In example 2, "ITMNO_INDICATOR" and "STOCKQTY_INDICATOR" are indicator variables. Refer to Table: Column values and values stored in indicator variables for details of the values stored in indicator variables by the FETCH statement.

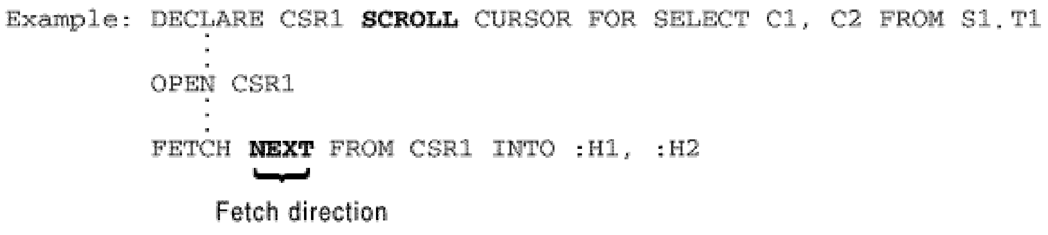
If indicator variables are not specified and fetched data contains a null value, the FETCH statement results in an error. Therefore, specify indicator variables if you do not know whether any of the values for columns are null values.

■Moving the cursor in different directions

The cursor can be moved in different directions by specifying a fetch direction in the FETCH statement. When specifying a fetch direction, specify SCROLL in the cursor declaration. When a fetch direction is omitted, NEXT is assumed to be specified.

The following are examples of specifying a fetch direction. The same cursor declaration applies to examples a) to f).

a) This example fetches the value of the row following the current row (NEXT specified). The cursor is positioned to [5] in the following Figure: Example of positioning a cursor in different directions.



```
Example: DECLARE CSR1 SCROLL CURSOR FOR SELECT C1, C2 FROM S1.T1
              :
         OPEN CSR1
              :
         FETCH NEXT FROM CSR1 INTO :H1, :H2
```

Fetch direction

b) This example fetches the value of the row preceding the current row (PRIOR specified). The cursor is positioned to [4] in the following Figure: Example of positioning a cursor in different directions.

```
Example: FETCH PRIOR FROM CSR1 INTO :H1, :H2
```

c) This example fetches the value of the first row (FIRST specified) regardless of the current row. The cursor is positioned to [1] in the following Figure: Example of positioning a cursor in different directions.

```
Example: FETCH FIRST FROM CSR1 INTO :H1, :H2
```

d) This example fetches the value of the last row (LAST specified) regardless of the current row. The cursor is positioned to [8] in the following Figure: Example of positioning a cursor in different directions.

```
Example: FETCH LAST FROM CSR1 INTO :H1, :H2
```

e) This example fetches the value of the nth row from the first or last row regardless of the current row (ABSOLUTE specified).

- The value of the second row from the first row is fetched. The cursor is positioned to [2] in the following Figure: Example of positioning a cursor in different directions.

```
Example: FETCH ABSOLUTE 2 FROM CSR1 INTO :H1, :H2
```

- The value of the second row from the last row is fetched. The cursor is positioned to [7] in the following Figure: Example of positioning a cursor in different directions.

```
Example: FETCH ABSOLUTE -2 FROM CSR1 INTO :H1, :H2
```

f) This example fetches the value of the nth row from the current row (RELATIVE specified).

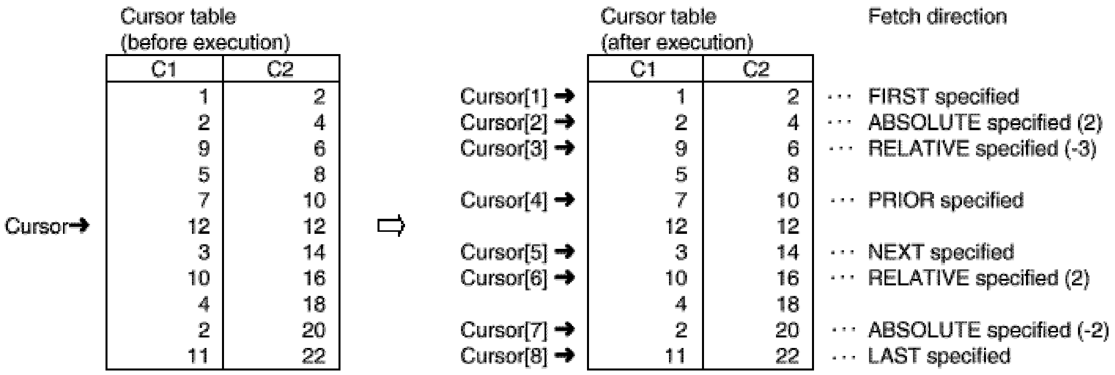
- The value of the second row down from the current row is fetched. The cursor is positioned to [6] in the following Figure: Example of positioning a cursor in different directions.

```
Example: FETCH RELATIVE 2 FROM CSR1 INTO :H1, :H2
```

- The value of the third row up from the current row is fetched. The cursor is positioned to [3] in the following Figure: Example of positioning a cursor in different directions.

```
Example: FETCH RELATIVE -3 FROM CSR1 INTO :H1, :H2
```

[Figure: Example of updating data using a cursor (column to be updated is in the cursor table)]


Cursor table
(before execution)
C1
C2
1 2
2 4
9 6
5 8
7 10
Cursor
12 12
3 14
10 16
4 18
2 20
11 22
Cursor table
(after execution)
Fetch direction
Cursor[1] 1 2 ··· FIRST specified
Cursor[2] 2 4 ··· ABSOLUTE specified (2)
Cursor[3] 9 6 ··· RELATIVE specified (-3)
5 8
Cursor[4] 7 10 ··· PRIOR specified
12 12
Cursor[5] 3 14 ··· NEXT specified
Cursor[6] 10 16 ··· RELATIVE specified (2)
4 18
Cursor[7] 2 20 ··· ABSOLUTE specified (-2)
Cursor[8] 11 22 ··· LAST specified

4.5 Updating Data Using a Cursor

Use an UPDATE statement to update data in a row in which a cursor is positioned. An updatable cursor must be used in an UPDATE statement. The following is an example of specifying an UPDATE statement:

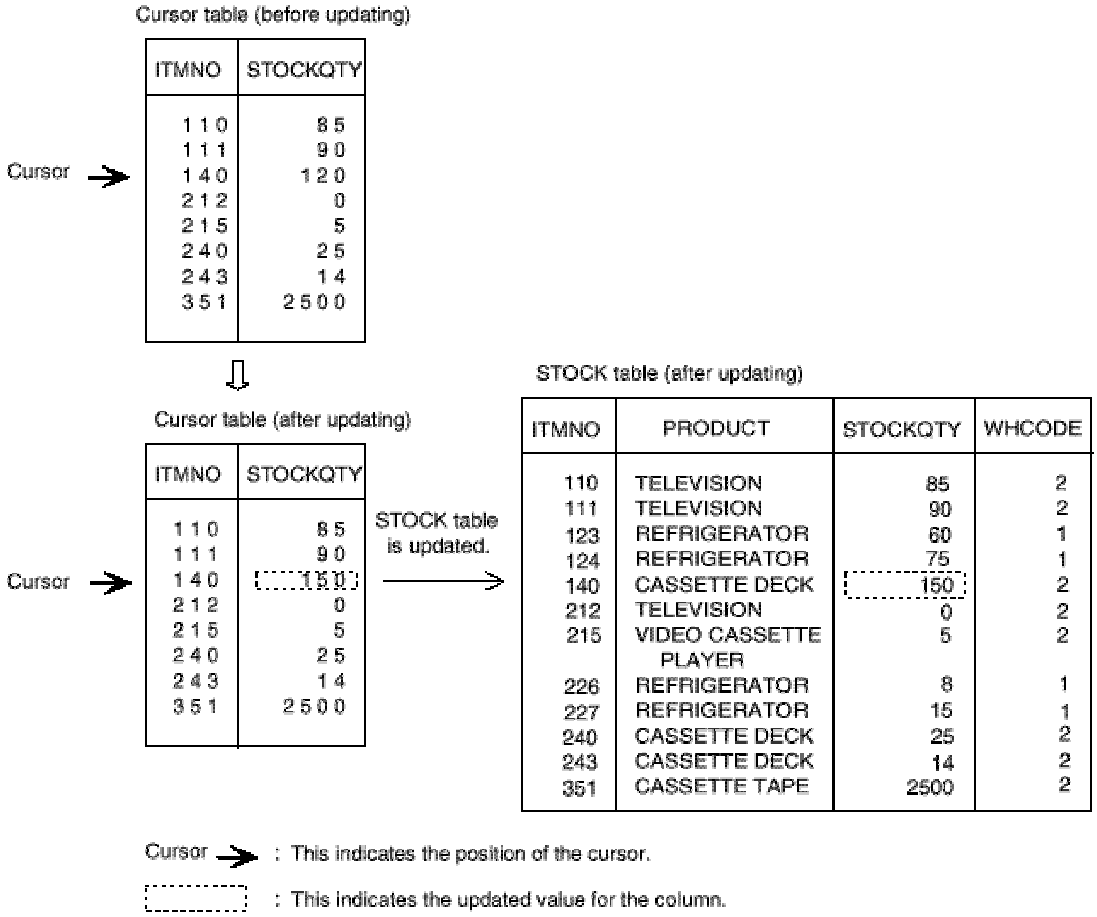
Example 1:

In this example, STOCKQTY is updated to "150" for the row in which the cursor is positioned in Figure: Example of using a FETCH statement to position a cursor.

$$\text{UPDATE }\underbrace{\text{STOCKS. STOCK}}_{\text{Table name}}\text{ SET }\underbrace{\text{STOCKQTY = 150}}_{\text{Set clause}}$$

$$\text{WHERE CURRENT OF }\underbrace{\text{STOCK\_CSR}}_{\text{Cursor name}}$$

[Figure: Example of updating data using a cursor (column to be updated is not in the cursor table)]


Cursor
STOCK table is updated.
Cursor
Cursor
: This indicates the position of the cursor.
: This indicates the updated value for the column.

Cursor table (before updating)

ITMNO	STOCKQTY
110	85
111	90
140	120
212	0
215	5
240	25
243	14
351	2500

Cursor table (after updating)

ITMNO	STOCKQTY
110	85
111	90
140	150
212	0
215	5
240	25
243	14
351	2500

STOCK table (after updating)

ITMNO	PRODUCT	STOCKQTY	WHCODE
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
140	CASSETTE DECK	150	2
212	TELEVISION	0	2
215	VIDEO CASSETTE PLAYER	5	2
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETTE TAPE	2500	2

Note: The cursor position before and after updating is the same.

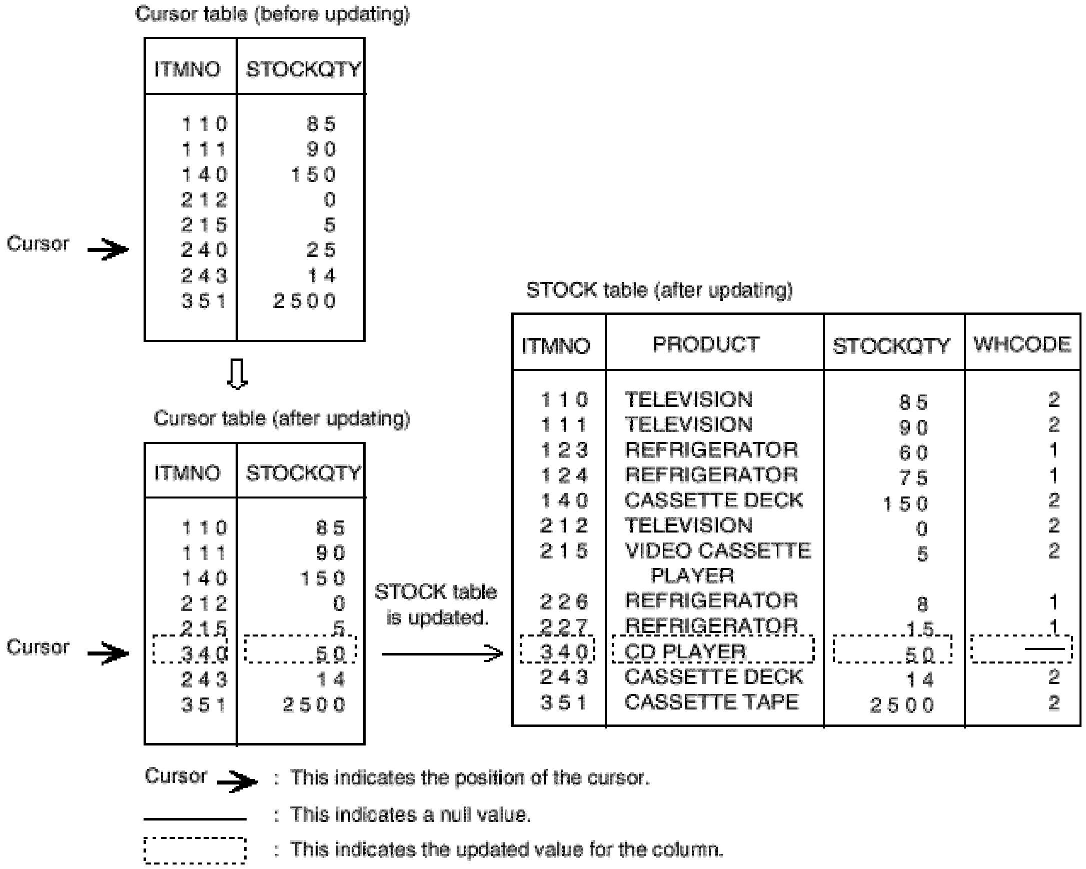
Data can also be updated using an UPDATE statement without using a cursor. For more information on this method, see 3.2 "Updating Data in a Data Base".

Example 2:

In this example, the cursor in Figure: Example of using a FETCH statement to position a cursor is moved again. Then ITMNO is updated to the value in the host variable ITMNO, PRODUCT is updated to "CD PLAYER", STOCKQTY is updated to twice its original value, and WHCODE is updated to a null value. Specify "340" in the host variable ITMNO.

```
UPDATE STOCKS. STOCK
    SET ITMNO = : ITMNO, PRODUCT = 'CD PLAYER',
        STOCKQTY = STOCKQTY * 2, WHCODE = NULL
WHERE CURRENT OF STOCK_CSR
```

[Figure: Example of updating data using a cursor (column to be updated is not in the curso r table)]



Cursor table (before updating)

ITMNO	STOCKQTY
110	85
111	90
140	150
212	0
215	5
240	25
243	14
351	2500

Cursor table (after updating)

ITMNO	STOCKQTY
110	85
111	90
140	150
212	0
215	5
340	50
243	14
351	2500

STOCK table (after updating)

ITMNO	PRODUCT	STOCKQTY	WHCODE
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
140	CASSETTE DECK	150	2
212	TELEVISION	0	2
215	VIDEO CASSETTE PLAYER	5	2
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
340	CD PLAYER	50	—
243	CASSETTE DECK	14	2
351	CASSETTE TAPE	2500	2

Cursor → : This indicates the position of the cursor.
——— : This indicates a null value.
┆┄┄┄┆ : This indicates the updated value for the column.

Note: The cursor position before and after updating is the same.

Data can also be updated using an UPDATE statement without using a cursor. For more information on this method, see 3.2 "Updating Data in a Data Base".

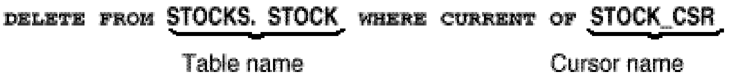
This method of updating data using an UPDATE statement is called UPDATE statement (positioned) because the row to be updated is specified by positioning the cursor.

4.6 Deleting Data Using a Cursor

Use the DELETE statement to delete data from a row in which a cursor is positioned. An updatable cursor must be specified in a DELETE statement. The following is an example of specifying the DELETE statement:

Example:

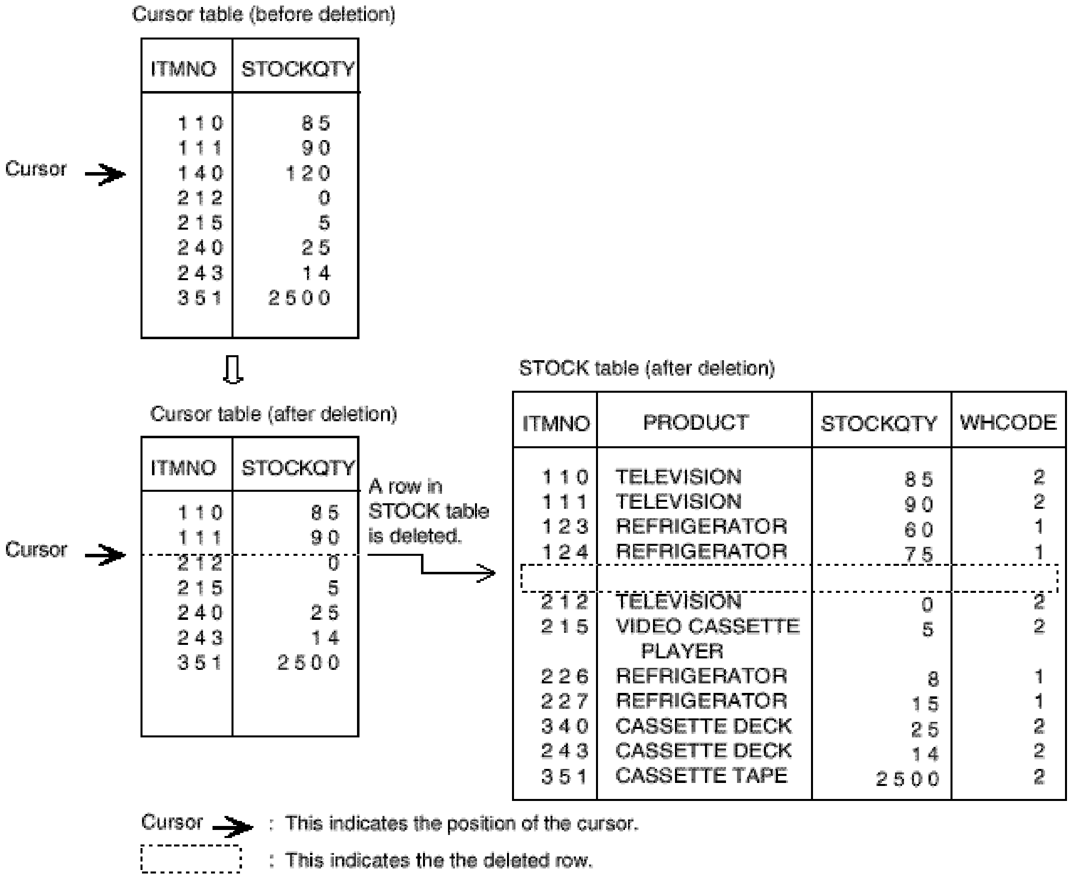
> In this example, the row in which the cursor is positioned in Figure: Example of using a FETCH statement to position a cursor is deleted.



```
DELETE FROM STOCKS.STOCK WHERE CURRENT OF STOCK_CSR
```

Table name
Cursor name

[Figure: Example of deleting data using a cursor]



Cursor table (before deletion)
Cursor
Cursor table (after deletion)
Cursor
A row in STOCK table is deleted.
STOCK table (after deletion)
Cursor → : This indicates the position of the cursor.
: This indicates the the deleted row.

Cursor table (before deletion)

ITMNO	STOCKQTY
110	85
111	90
140	120
212	0
215	5
240	25
243	14
351	2500

Cursor table (after deletion)

ITMNO	STOCKQTY
110	85
111	90
212	0
215	5
240	25
243	14
351	2500

STOCK table (after deletion)

ITMNO	PRODUCT	STOCKQTY	WHCODE
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
212	TELEVISION	0	2
215	VIDEO CASSETTE PLAYER	5	2
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
340	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETTE TAPE	2500	2

Note: The cursor is positioned between rows after a row is deleted.

Data can also be deleted using an UPDATE statement without using a cursor. For more information on this method, see 3.3 "Deleting Data from a Data Base".

This method of deleting data using a DELETE statement is called DELETE statement (positioned) because the row to be deleted is specified by positioning the cursor.

Chapter 5 Joining Multiple Tables and Manipulating data

This chapter describes how to join multiple tables and manipulate data when data is fetched from data bases. For example, suppose that to check on the status of orders for products, data for CUSTOMER, PRODUCT, and ORDERQTY is fetched from the stock inventory data base. This data is to be fetched from the ORDER table, but data for PRODUCT is discovered to be unavailable there. However, PRODUCT data is available in the STOCK table. As shown in Figure: Relationship between STOCK table, ORDER table, and COMPANY table, the STOCK table and the ORDER table are related through ITMNO in the STOCK table and PRODNO in the ORDER table. Therefore, these two tables can be joined and data fetched.

5.1 Deriving a New Table from Multiple Tables

5.2 Specifying Conditions to Join Tables

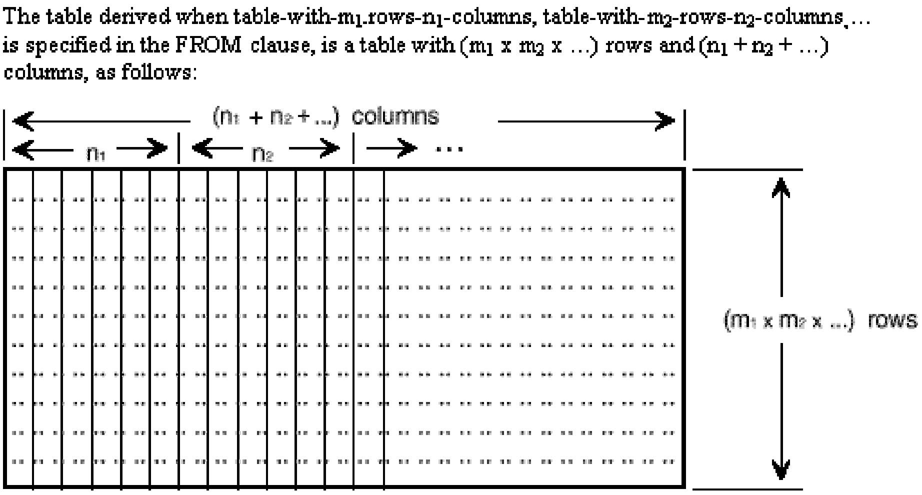
5.3 Manipulating Data Using Aliases in Tables

5.4 Joining a Table to Itself and Manipulating Data

5.5 Obtaining the Aggregate for Rows from Multiple Tables

5.1 Deriving a New Table from Multiple tables

To join multiple tables and fetch data, specify the table names, separated by commas, in the FROM clause of the table expression. When multiple table names are specified, a new table is derived from the result of the FROM clause. Data can then be fetched from the derived table. The kind of table derived when multiple table names are specified is as follows:



The table derived when table-with-m_1.rows-n_1-columns, table-with-m_2-rows-n_2-columns,... is specified in the FROM clause, is a table with (m_1 x m_2 x ...) rows and (n_1 + n_2 + ...) columns, as follows:

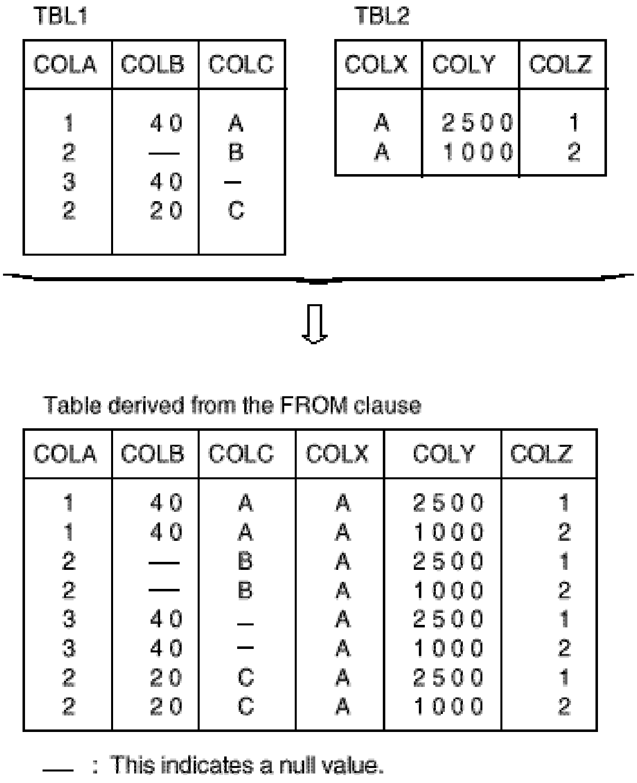
The following is an example of fetching data from multiple tables:

Example:

In this example, table names TBL1 and TBL2 are specified in the FROM clause and SCM is the schema name for these tables.

```
SELECT  ...  FROM SCM.TBL1, SCM.TBL2
```

[Figure: Table derived when two table names are specified in a FROM clause]



TBL1

COLA	COLB	COLC
1	40	A
2	—	B
3	40	—
2	20	C

TBL2

COLX	COLY	COLZ
A	2500	1
A	1000	2

Table derived from the FROM clause

COLA	COLB	COLC	COLX	COLY	COLZ
1	40	A	A	2500	1
1	40	A	A	1000	2
2	—	B	A	2500	1
2	—	B	A	1000	2
3	40	—	A	2500	1
3	40	—	A	1000	2
2	20	C	A	2500	1
2	20	C	A	1000	2

— : This indicates a null value.

If two table names are specified in a FROM clause, the table derived from the FROM clause contains all columns and rows in the two specified tables. This is called the expanded direct product of the table. In the same way, if three or more tables names are specified in the FROM clause, the derived table contains all columns and rows in the specified tables. In other words, the number of columns that results is the summation of the number of columns in the specified tables. The row count is the product of the row counts of the specified tables. The sequence of columns in the derived table is the sequence of columns in each table in the order in which the tables were specified.

5.2 Specifying Conditions to Join Tables

The expanded direct product derived from specifying multiple tables in a FROM clause combines all rows from the specified tables, although many of the rows may contain insignificant data. When this data is processed by an application program, all the rows are fetched, but the program only processes rows containing significant data. Because a large number of rows are fetched, processing is not very efficient. So, in addition to joining tables, data must be narrowed down to just significant data. To do so, specify a condition to join tables in the WHERE clause of the table expression. The following is an example of specifying conditions in the WHERE clause:

Example 1:

In this example, the data that joins the two tables TBL1 and TBL2 in the example in Figure: Table derived when two table names are specified in a FROM clause is specified as values for column COLA and column COLZ.

```
SELECT  ...  FROM SCM.TBL1, SCM.TBL2  WHERE COLA = COLZ
```

[Figure: Example of specifying conditions in WHERE clause for joining tables]

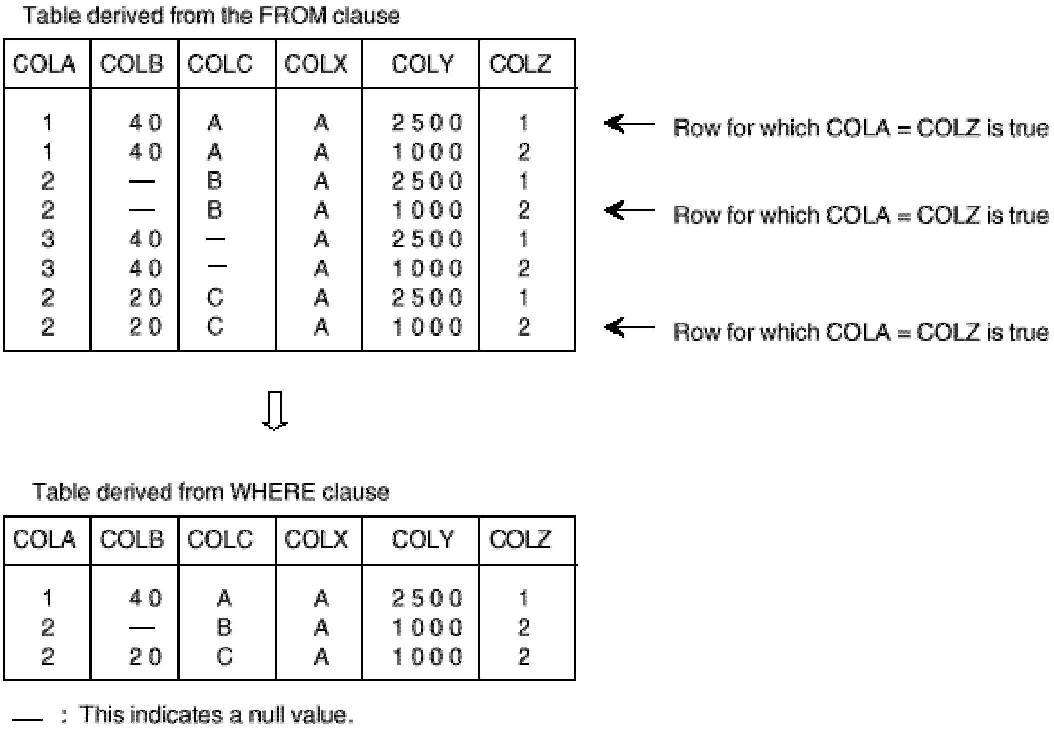


Table derived from the FROM clause

COLA	COLB	COLC	COLX	COLY	COLZ
1	40	A	A	2500	1
1	40	A	A	1000	2
2	—	B	A	2500	1
2	—	B	A	1000	2
3	40	—	A	2500	1
3	40	—	A	1000	2
2	20	C	A	2500	1
2	20	C	A	1000	2

Row for which COLA = COLZ is true

Row for which COLA = COLZ is true

Row for which COLA = COLZ is true

Table derived from WHERE clause

COLA	COLB	COLC	COLX	COLY	COLZ
1	40	A	A	2500	1
2	—	B	A	1000	2
2	20	C	A	1000	2

— : This indicates a null value.

In Example 1, because the "COLA = COLZ" condition is specified in the WHERE clause, the only rows fetched are those in which the values of columns COLA and COLZ are the same.

When multiple tables are specified in a FROM clause, columns with the same name may exist in more than one table. To specify a column name that may exist in more than one table, qualify the column name by preceding it with a table name and a period. For example, if the column name COLA exists in both tables TBL1 and TBL2, specify column COLA in table TBL1 as "TBL1.COLA" or "SCM.TBL1.COLA." Similarly, specify column COLA in table TBL2 as "TBL2.COLA" or "SCM.TBL2.COLA".

As an example of joining multiple tables, the following examines the status of orders mentioned at the beginning of this chapter.

Example 2:

> In this example, CUSTOMER, PRODUCT, and ORDERQTY are fetched from the ORDER table and the STOCK table. The column names are specified with a table name qualifier. (In reality, the ORDER table and the STOCK table do not contain columns with the same name, so table name qualifiers need not be specified.) Table name qualifiers are only required when multiple tables specified in a FROM clause contain columns with the same name.

```
SELECT  ORDER. CUSTOMER,  STOCK. PRODUCT,  ORDER. ORDERQTY

       FROM  STOCKS. ORDER,  STOCKS. STOCK

       WHERE  ORDER. PRODNO =  STOCK. ITMNO
```

[Figure: Example in which multiple tables are joined and data manipulated]

ORDER table

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	123	48000	60
61	124	64000	40
61	140	8000	80
61	215	240000	10
61	240	80000	20
62	110	37500	120
62	226	112500	20
62	351	375	800
63	111	57400	80
63	212	205000	30
63	215	246000	10
71	140	7800	50
71	351	390	600
72	140	7000	70
72	215	210000	10
72	226	105000	20
72	243	84000	10
72	351	350	1000
74	110	39000	120
74	111	54000	120
74	226	117000	20
74	227	140400	10
74	351	390	700

STOCK table

ITMNO	PRODUCT	STOCKQTY	WHCODE
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
140	CASSETTE DECK	120	2
212	TELEVISION	0	2
215	VIDEO CASSETTE PLAYER	5	2
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETTE TAPE	2500	2

⇩

Table derived from FROM clause

CUSTOMER	PRODNO	PRICE	ORDERQTY	ITMNO	PRODUCT	STOCKQTY	WHCODE
61	123	48000	60	110	TELEVISION	85	2
61	123	48000	60	111	TELEVISION	90	2
61	123	48000	60	123	REFRIGERATOR	60	1
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.

(The table derived from the FROM clause contains 276 rows, which is a combination of all rows from ORDER table and the STOCK table.)

⇩

Table derived from WHERE clause

CUSTOMER	PRODNO	PRICE	ORDERQTY	ITMNO	PRODUCT	STOCKQTY	WHCODE
61	123	48000	60	123	REFRIGERATOR	60	1
61	124	64000	40	124	REFRIGERATOR	75	1
61	140	8000	80	140	CASSETTE DECK	120	2
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.

(The table derived from the WHERE clause contains 23 rows. These are the rows in which the values of PRODNO and ITMNO are equal in the table derived from the FROM clause.)

⇩

Table derived from query specification

CUSTOMER	PRODUCT	ORDERQTY
61	REFRIGERATOR	60
61	REFRIGERATOR	40
61	CASSETTE DECK	80
61	VIDEO CASSETTE PLAYER	10
61	CASSETTE DECK	20
62	TELEVISION	120
62	REFRIGERATOR	20
62	CASSETTE TAPE	800
63	TELEVISION	80
63	TELEVISION	30
63	VIDEO CASSETTE PLAYER	10
71	CASSETE DECK	50
71	CASSETTE TAPE	600
72	CASSETTE DECK	70
72	VIDEO CASSETTE PLAYER	10
72	REFRIGERATOR	20
72	CASSETTE DECK	10
72	CASSETTE TAPE	1000
74	TELEVISION	120
74	TELEVISION	120
74	REFRIGERATOR	20
74	REFRIGERATOR	10
74	CASSETTE TAPE	700

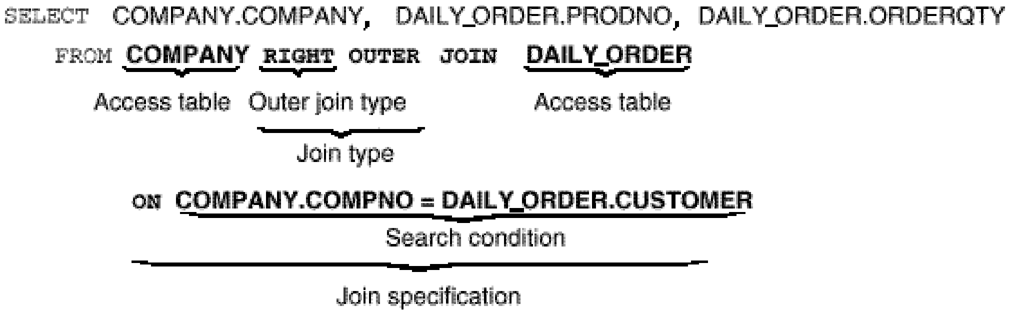
■Join tables

In Example 2, all rows in the ORDER table and STOCK table are in the table derived from the WHERE clause. Some rows that are not in the ORDER or STOCK table may also be present depending on the conditions. To also fetch rows that do not satisfy the conditions, use a join table. A join table can be used to also fetch rows which do not satisfy the join specification fetch conditions, but which are in the tables specified by the join type. When LEFT is coded as the outer join type, all rows of the table to the left of the join type are fetched. When RIGHT is coded, all rows of the table on the right are fetched. A specification example is given below.

Example 3:

> In this example, COMPANY, PRODNO, and ORDERQTY are fetched from the COMPANY table and DAILY_ORDER table for the data in which the values of COMPNO and CUSTOMER are equal. All rows in the DAILY_ORDER table that have different values for COMPNO and CUSTOMER are also

fetched.



```
SELECT  COMPANY.COMPANY,  DAILY_ORDER.PRODNO,  DAILY_ORDER.ORDERQTY
   FROM COMPANY RIGHT OUTER JOIN  DAILY_ORDER
        ON COMPANY.COMPNO = DAILY_ORDER.CUSTOMER
```

[Figure: Example in which multiple tables are joined using a join table]

COMPANY table

COMPNO	COMPANY	PHONE	ADDRESS
61	IDEA INC.	43	LONDON
62	ADAM LTD.	73	SANTA CLARA
63	MOON CO.	14	FIFTH AVENUE
71	RIVER CO.	34	SAKAI OSAKA
72	DRAGON CO.	37	HARAJUKU
74	FIRST CO.	25	SYDNEY

DAILY_ORDER table

CUSTOMER	PRODNO	ORDERQTY	PROCESS_TIME
74	124	10	09:40:33
61	351	100	10:26:05
59	111	5	13:01:58
63	240	12	15:54:20
63	351	500	15:55:49

⇩

Table derived by combining all columns and rows of the COMPANY table and DAILY_ORDER table

DAILY_ORDER table

COMPNO	COMPANY	PHONE	ADDRESS	CUSTOMER	PRODNO	ORDERQTY	PROCESS_TIME
61	IDEA LTD.	43	LONDON	74	124	10	09:40:33
61	IDEA LTD.	43	LONDON	61	351	100	10:26:05
61	IDEA LTD.	43	LONDON	59	111	5	13:01:58
61	IDEA LTD.	43	LONDON	63	240	12	15:54:20
61	IDEA LTD.	43	LONDON	63	351	500	15:55:49
62	ADAM INC.	73	SANTA CLARA	74	124	10	09:40:33
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.

(The table derived by combining all columns and rows of the COMPANY table and DAILY_ORDER table contains 30 rows.)

⇩

Table derived by specifying join

DAILY_ORDER table

COMPNO	COMPANY	PHONE	ADDRESS	CUSTOMER	PRODNO	ORDERQTY	PROCESS_TIME
61	IDEA INC.	43	LONDON	61	351	100	10:26:05
63	MOON CO.	14	FIFTH AVENUE	63	240	12	15:54:20
63	MOON CO.	14	FIFTH AVENUE	63	351	500	15:55:49
74	FIRST CO.	25	SYDNEY	74	124	10	09:40:33

⇩

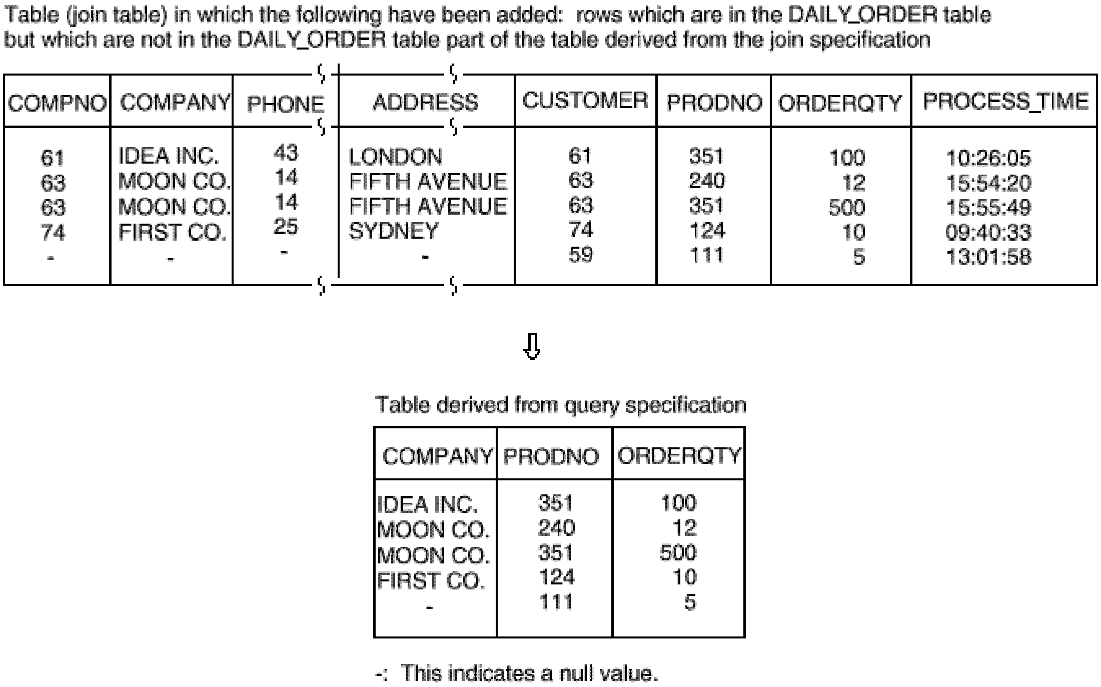


Table (join table) in which the following have been added: rows which are in the DAILY_ORDER table but which are not in the DAILY_ORDER table part of the table derived from the join specification

COMPNO	COMPANY	PHONE	ADDRESS	CUSTOMER	PRODNO	ORDERQTY	PROCESS_TIME
61	IDEA INC.	43	LONDON	61	351	100	10:26:05
63	MOON CO.	14	FIFTH AVENUE	63	240	12	15:54:20
63	MOON CO.	14	FIFTH AVENUE	63	351	500	15:55:49
74	FIRST CO.	25	SYDNEY	74	124	10	09:40:33
-	-	-	-	59	111	5	13:01:58

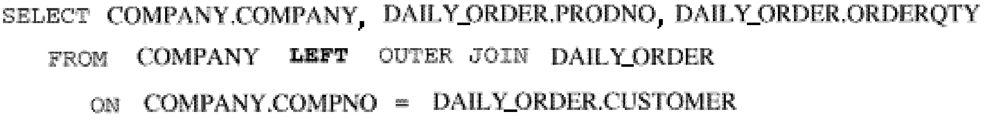
⇩

Table derived from query specification

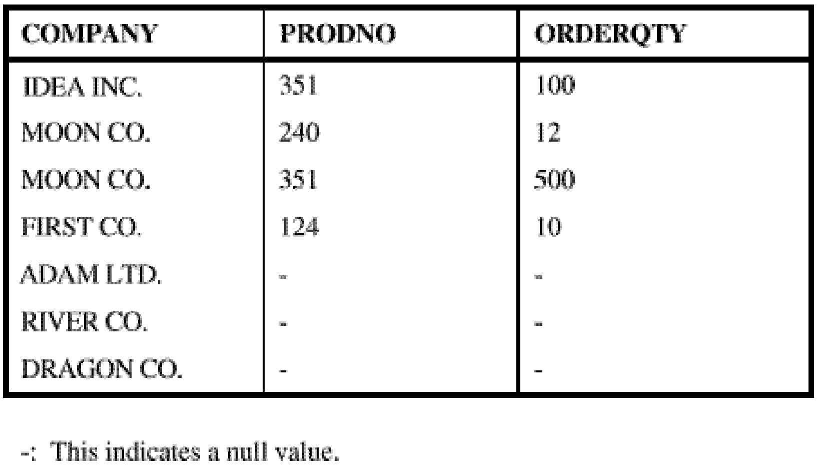
COMPANY	PRODNO	ORDERQTY
IDEA INC.	351	100
MOON CO.	240	12
MOON CO.	351	500
FIRST CO.	124	10
-	111	5

-: This indicates a null value.

Example 3 illustrates coding RIGHT for the outer join type. If LEFT is coded for the outer join type in the same example statements, the results are as follows:



```
SELECT  COMPANY.COMPANY, DAILY_ORDER.PRODNO, DAILY_ORDER.ORDERQTY
   FROM  COMPANY  LEFT  OUTER JOIN  DAILY_ORDER
      ON  COMPANY.COMPNO  =  DAILY_ORDER.CUSTOMER
```



COMPANY	PRODNO	ORDERQTY
IDEA INC.	351	100
MOON CO.	240	12
MOON CO.	351	500
FIRST CO.	124	10
ADAM LTD.	-	-
RIVER CO.	-	-
DRAGON CO.	-	-

-: This indicates a null value.

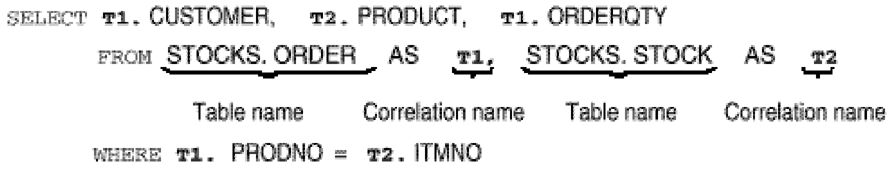
5.3 Manipulating Data Using Aliases in Tables

If column names are qualified by long table names, coding columns may become time-consuming. Instead, data can be manipulated by specifying correlation names for tables in table expressions. The following is an example of specifying correlation names:

Example:

> In this example, correlation names are used as column name qualifiers in the query specification in Figure: Example in which multiple tables are joined and data manipulated. The result of the query specification is the same as in Figure: Example in which multiple tables are joined and data

manipulated.



```
SELECT T1. CUSTOMER,   T2. PRODUCT,   T1. ORDERQTY
       FROM STOCKS. ORDER  AS  T1,  STOCKS. STOCK  AS  T2
       WHERE T1. PRODNO = T2. ITMNO
```

In this example, the correlation names "T1" and "T2" are specified for the ORDER table and the STOCK table, respectively. One of these correlation names is used as the column name qualifier. The correlation name is valid only in the specified SQL statement. If a correlation name is specified, the correlation name must be used in the column name qualifier. The original table name cannot be used as the column name qualifier.

5.4 Joining a Table to Itself and Manipulating Data

This manual has already explained comparing a value in one column with a value in another column and performing operations on values in rows of columns in tables. Comparing and manipulating values in different rows of the same table has not so far been possible. This section explains how to manipulate data in different rows by joining a table to itself as if joining it to a different table.

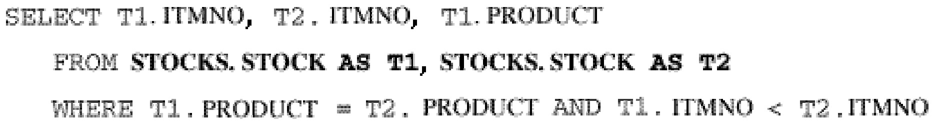
To join a table to itself, specify the table name more than once in the FROM clause. Then, specify a correlation name in the SQL statement for each table so that the tables can be distinguished.

■Comparing different rows

The following is an example of comparing values from different rows of the same table:

Example 1:

In this example, pairs of ITMNO for which PRODUCT is identical are obtained from the STOCK table. The columns to be fetched are the ITMNO pair and PRODUCT. To perform this data manipulation, the STOCK table is specified twice in the FROM clause and a table is derived that includes two identical columns. To distinguish identical columns, add the correlation name T1 to one column and T2 to the other.



```
SELECT T1.ITMNO, T2. ITMNO, T1.PRODUCT
   FROM STOCKS.STOCK AS T1, STOCKS.STOCK AS T2
   WHERE T1.PRODUCT = T2. PRODUCT AND T1. ITMNO < T2.ITMNO
```

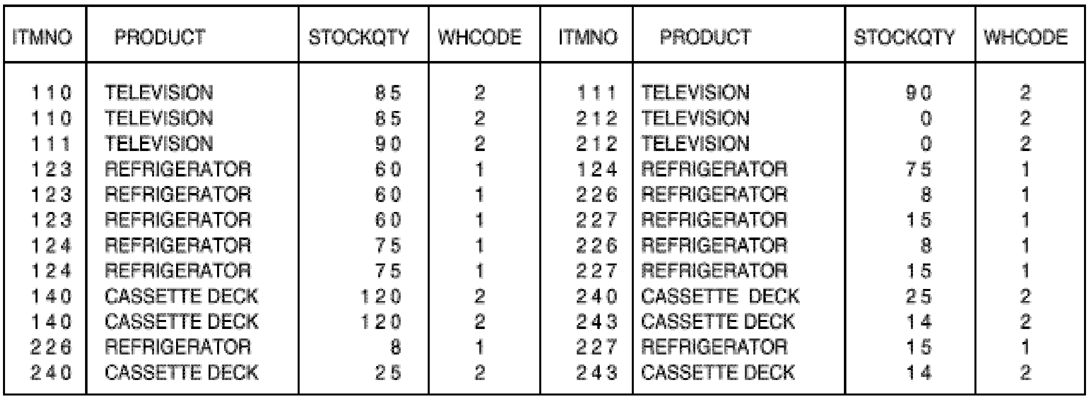
[Figure: Example of joining a table to itself and manipulating data]

Table derived from FROM clause

T1				T2			
ITMNO	PRODUCT	STOCKQTY	WHCODE	ITMNO	PRODUCT	STOCKQTY	WHCODE
110	TELEVISION	85	2	110	TELEVISION	85	2
110	TELEVISION	85	2	111	TELEVISION	90	2
110	TELEVISION	85	2	123	REFRIGERATOR	60	1
110	TELEVISION	85	2	124	REFRIGERATOR	75	1
110	TELEVISION	85	2	140	CASSETTE DECK	120	2
110	TELEVISION	85	2	212	TELEVISION	0	2
110	TELEVISION	85	2	215	VIDEO CASSETTE PLAYER	5	2
110	TELEVISION	85	2	226	REFRIGERATOR	8	1
110	TELEVISION	85	2	227	REFRIGERATOR	15	1
110	TELEVISION	85	2	240	CASSETTE DECK	25	2
110	TELEVISION	85	2	243	CASSETTE DECK	14	2
110	TELEVISION	85	2	351	CASSETTE TAPE	2500	2
111	TELEVISION	90	2	110	TELEVISION	85	2
111	TELEVISION	90	2	111	TELEVISION	90	2
111	TELEVISION	90	2	123	REFRIGERATOR	60	1
111	TELEVISION	90	2	124	REFRIGERATOR	75	1
111	TELEVISION	90	2	140	CASSETTE DECK	120	2
111	TELEVISION	90	2	212	TELEVISION	0	2
111	TELEVISION	90	2	215	VIDEO CASSETTE PLAYER	5	2
111	TELEVISION	90	2	226	REFRIGERATOR	8	1
111	TELEVISION	90	2	227	REFRIGERATOR	15	1
111	TELEVISION	90	2	240	CASSETTE DECK	25	2
111	TELEVISION	90	2	243	CASSETTE DECK	14	2
111	TELEVISION	90	2	351	CASSETTE TAPE	2500	2
123	REFRIGERATOR	60	1	110	TELEVISION	85	2
123	REFRIGERATOR	60	1	111	TELEVISION	90	2
123	REFRIGERATOR	60	1	123	REFRIGERATOR	60	1
123	REFRIGERATOR	60	1	124	REFRIGERATOR	75	1
123	REFRIGERATOR	60	1	140	CASSETTE DECK	120	2
123	REFRIGERATOR	60	1	212	TELEVISION	0	2
123	REFRIGERATOR	60	1	215	VIDEO CASSETTE PLAYER	5	2
123	REFRIGERATOR	60	1	226	REFRIGERATOR	8	1
123	REFRIGERATOR	60	1	227	REFRIGERATOR	15	1
123	REFRIGERATOR	60	1	240	CASSETTE DECK	25	2
123	REFRIGERATOR	60	1	243	CASSETTE DECK	14	2
123	REFRIGERATOR	60	1	351	CASSETTE TAPE	2500	2
351	CASSETTE TAPE	2500	2	227	REFRIGERATOR	15	1
351	CASSETTE TAPE	2500	2	240	CASSETTE DECK	25	2
351	CASSETTE TAPE	2500	2	243	CASSETTE DECK	14	2
351	CASSETTE TAPE	2500	2	351	CASSETTE TAPE	2500	2

⇩

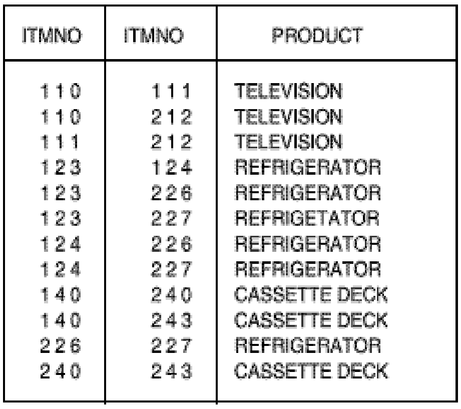
Table derived from WHERE clause



ITMNO	PRODUCT	STOCKQTY	WHCODE	ITMNO	PRODUCT	STOCKQTY	WHCODE
110	TELEVISION	85	2	111	TELEVISION	90	2
110	TELEVISION	85	2	212	TELEVISION	0	2
111	TELEVISION	90	2	212	TELEVISION	0	2
123	REFRIGERATOR	60	1	124	REFRIGERATOR	75	1
123	REFRIGERATOR	60	1	226	REFRIGERATOR	8	1
123	REFRIGERATOR	60	1	227	REFRIGERATOR	15	1
124	REFRIGERATOR	75	1	226	REFRIGERATOR	8	1
124	REFRIGERATOR	75	1	227	REFRIGERATOR	15	1
140	CASSETTE DECK	120	2	240	CASSETTE DECK	25	2
140	CASSETTE DECK	120	2	243	CASSETTE DECK	14	2
226	REFRIGERATOR	8	1	227	REFRIGERATOR	15	1
240	CASSETTE DECK	25	2	243	CASSETTE DECK	14	2

⇩

Table derived from query specification



ITMNO	ITMNO	PRODUCT
110	111	TELEVISION
110	212	TELEVISION
111	212	TELEVISION
123	124	REFRIGERATOR
123	226	REFRIGERATOR
123	227	REFRIGETATOR
124	226	REFRIGERATOR
124	227	REFRIGERATOR
140	240	CASSETTE DECK
140	243	CASSETTE DECK
226	227	REFRIGERATOR
240	243	CASSETTE DECK

■Operations on different rows

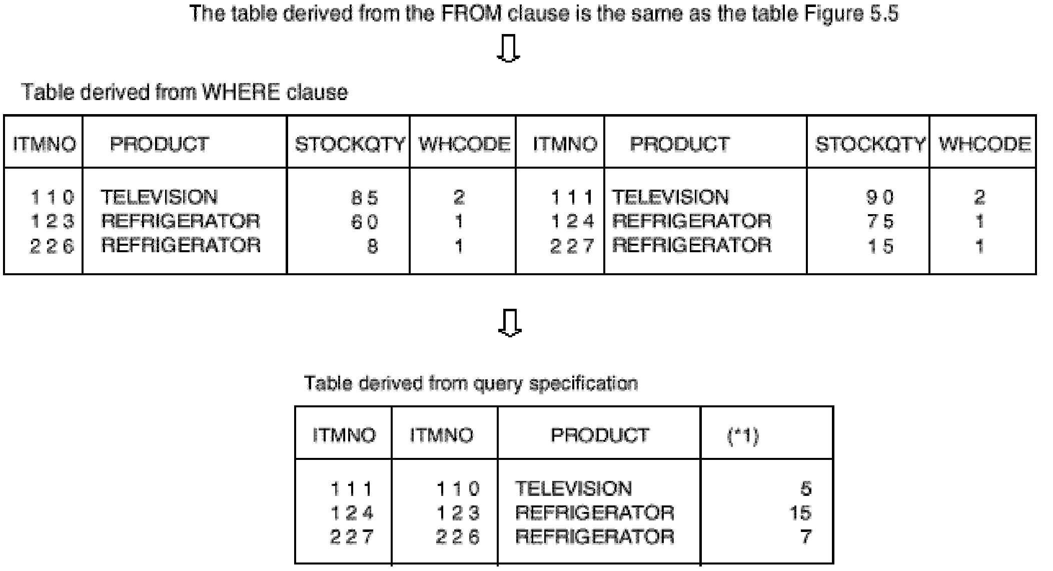
The following is an example of performing an operation on values in different rows of the same table:

Example 2:

In this example, the difference in STOCKQTY for products for which ITMNO is consecutive in the STOCK table is obtained. The data to be fetched is ITMNO for both products, PRODUCT for the one with the higher ITMNO, and the difference in STOCKQTY.

```
SELECT T2. ITMNO,  T1. ITMNO,  T2.PRODUCT,  T2.STOCKQTY - T1.STOCKQTY AS STOCKQTYDIFER

       FROM STOCKS. STOCK AS T1,  STOCKS. STOCK AS T2

       WHERE T2. ITMNO = T1. ITMNO + 1
```

[Figure: Example of performing an operation on different rows]



The table derived from the FROM clause is the same as the table Figure 5.5

⇩

Table derived from WHERE clause

ITMNO	PRODUCT	STOCKQTY	WHCODE	ITMNO	PRODUCT	STOCKQTY	WHCODE
110	TELEVISION	85	2	111	TELEVISION	90	2
123	REFRIGERATOR	60	1	124	REFRIGERATOR	75	1
226	REFRIGERATOR	8	1	227	REFRIGERATOR	15	1

⇩

Table derived from query specification

ITMNO	ITMNO	PRODUCT	(*1)
111	110	TELEVISION	5
124	123	REFRIGERATOR	15
227	226	REFRIGERATOR	7

*1 The column for the result of the operational expression does not have a name. This column contains the result of "T2.STOCKQTY - T1.STOCK QTY."

5.5 Obtaining the Aggregate for Rows from Multiple Tables

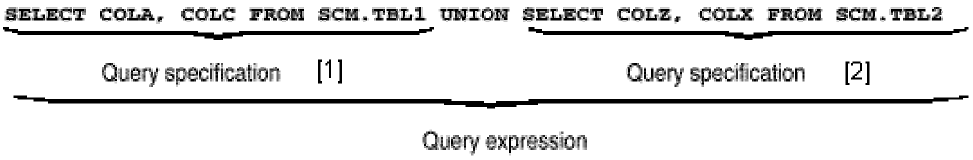
The aggregate for rows contained in multiple tables can be obtained by joining two or more query specifications with "UNION". Specify the tables to be processed and the conditions for fetching data in the query specifications. In obtaining the aggregate, rows in which the column values are the same can be specified as one row. Also, the original row count can be specified without modification.

■Specifying duplicate rows as one row during aggregation

The result of using UNION to join query specifications is that the rows for each query specification result are collected and a table is derived. In addition, duplicate rows are specified as one row. Duplicate rows are those rows in which the column values are the same. Figure: Example of specifying UNION in query expression shows the kind of tables that are derived from the result of using UNION to join query specifications.

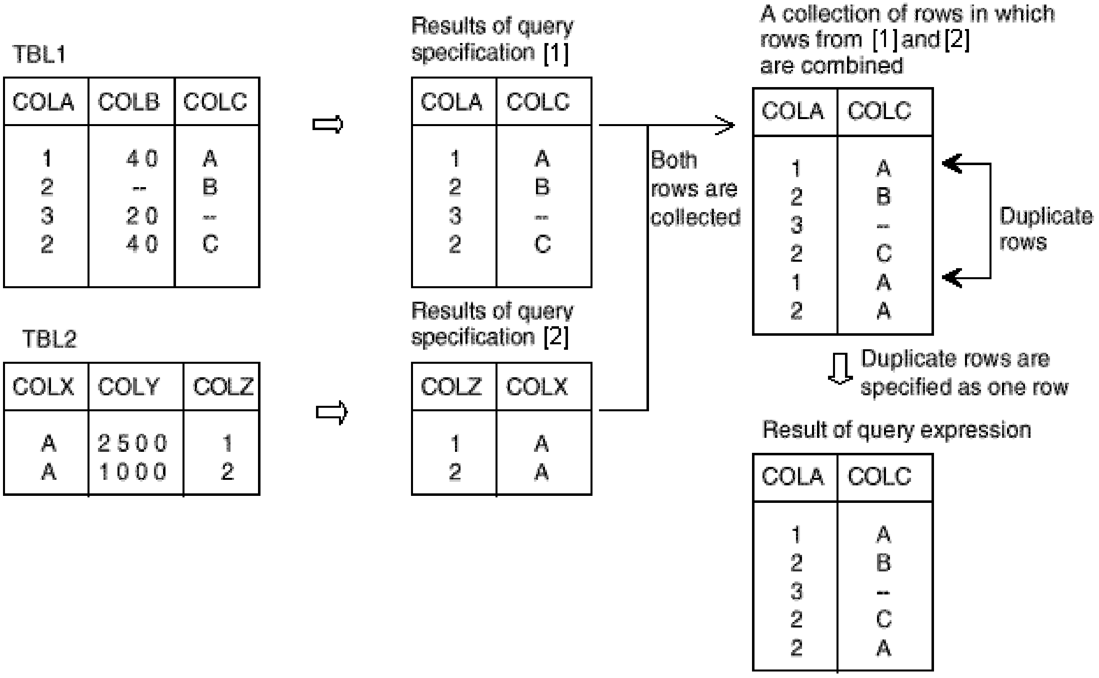
Example 1

> In this example, the aggregate is obtained for rows in columns COLA and COLC in table TBL1 and columns COLZ and COLX in table TBL2 in Figure: Example of specifying UNION in query expression. Tables TBL1 and TBL2 are contained in schema SCM.



```
SELECT COLA, COLC FROM SCM.TBL1 UNION SELECT COLZ, COLX FROM SCM.TBL2
```

[Figure: Example of specifying UNION in query expression]


TBL1
TBL2
Results of query specification [1]
Results of query specification [2]
Both rows are collected
A collection of rows in which rows from [1] and [2] are combined
Duplicate rows
Duplicate rows are specified as one row
Result of query expression

TBL1

COLA	COLB	COLC
1	40	A
2	--	B
3	20	--
2	40	C

TBL2

COLX	COLY	COLZ
A	2500	1
A	1000	2

Results of query specification [1]

COLA	COLC
1	A
2	B
3	--
2	C

Results of query specification [2]

COLZ	COLX
1	A
2	A

A collection of rows in which rows from [1] and [2] are combined

COLA	COLC
1	A
2	B
3	--
2	C
1	A
2	A

Result of query expression

COLA	COLC
1	A
2	B
3	--
2	C
2	A

--: Indicates a null value

If UNION is used to join query specifications, a new table is derived corresponding to the columns in both query specification results. The column sequence is from the left. Each query specification must be specified as follows:

- The number of columns in the results table must be the same as in the query specification.
- The data type for columns for each corresponding results table must be comparable.

For information about comparable data types, see Table "Comparable data types".

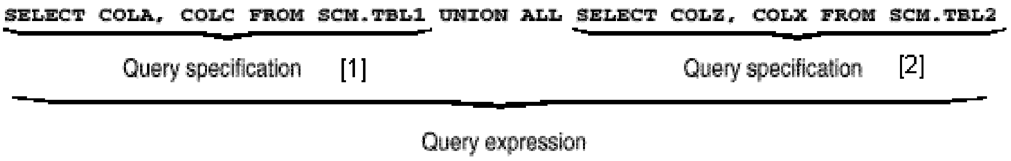
The column names in the table for the query expression results are the column names specified in the query specification on the left. However, if the select column list in the query specification on the left is an operational expression or set function, the columns in the results table do not have names.

■Including duplicate rows in aggregate without modification

Specify ALL and UNION to derive a table from all rows from each query specification result without having duplicate rows specified as one row. Figure: Example of specifying UNION ALL in query specification shows the kind of table derived when ALL and UNION are specified.

Example 2:

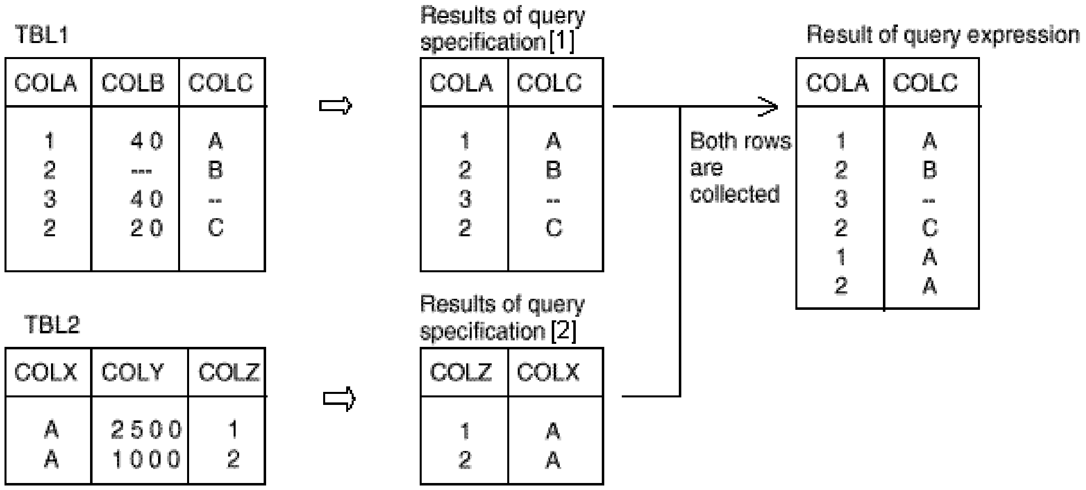
In this example, the aggregate is obtained for all rows in columns COLA and COLC in table TBL1 and columns COLZ and COLX in table TBL2 in Figure: Example of specifying UNION ALL in query specification. Duplicate rows are also included.



```
SELECT COLA, COLC FROM SCM.TBL1 UNION ALL SELECT COLZ, COLX FROM SCM.TBL2
```

Query specification [1]
Query specification [2]
Query expression

[Figure: Example of specifying UNION ALL in query specification]



TBL1

COLA	COLB	COLC
1	40	A
2	---	B
3	40	--
2	20	C

TBL2

COLX	COLY	COLZ
A	2500	1
A	1000	2

Results of query specification[1]

COLA	COLC
1	A
2	B
3	--
2	C

Results of query specification [2]

COLZ	COLX
1	A
2	A

Result of query expression

COLA	COLC
1	A
2	B
3	--
2	C
1	A
2	A

■Aggregate sequence due to UNION

Query specifications joined by UNION are processed in sequence from the left. However, parentheses can be used to specify a different evaluation sequence. A query expression enclosed in parentheses is processed before other query expressions. The following is an example of using parentheses in a query specification joined by UNION:

Example 3:

In this example, an aggregate is obtained for tables TBL1, TBL2, TBL3, and TBL4. First, duplicate rows in tables TBL1 and TBL2 are removed and the aggregate is obtained [1]. Next, duplicate rows in tables TBL3 and TBL4 are removed and the aggregate is obtained [2]. Finally, all rows, including the duplicated rows, in these results are included and the aggregate is obtained [3].

```
(SELECT COLA FROM SCM.TBL1 UNION SELECT COLZ FROM SCM.TBL2)
                           [1]
    UNION ALL
       [3]
 (SELECT COL1 FROM SCM.TBL3 UNION SELECT COL1 FROM SCM.TBL4)
                            [2]
```

Chapter 6 Methods of Manipulating Data

This chapter describes methods of using SQL statements to manipulate data.

This chapter contains the following sections:

6.1 Grouping Tables and Manipulating Data

6.2 Specifying Various Search Conditions

6.3 Manipulating Numeric Data

6.4 Manipulating Character String Data

6.5 Manipulating Date Data

6.6 Converting the Data Type to Manipulate Data

6.7 Using CASE Expression to Manipulate Data

6.8 Specifying Row IDs to Manipulate Data

6.9 Manipulating Data using Parallel Query

6.10 Omitting Schema Names

6.11 Changing the User of the Current Session

6.12 Adding a Name to a Results Column in Query Specification

6.13 Manipulating Data Using Sequence

6.1 Grouping Tables and Manipulating Data

When data is fetched from tables, the table can be grouped according to column values and data can be manipulated. Grouping tables refers to a way of grouping rows in which a column or a group of columns have the same value. A different group is formed from different rows. A table that has been grouped in this way is called a grouped table.

Specify the GROUP BY clause or the HAVING clause to group a table and manipulate data. The GROUP BY clause and the HAVING clause can be specified in a table expression. If the FROM clause, WHERE clause, GROUP BY clause, and HAVING clause are specified in the same table expression, the processing sequence is as follows:

1. FROM clause
2. WHERE clause
3. GROUP BY clause
4. HAVING clause

First, the rows in the table specified by the FROM clause are specified by the search conditions in the WHERE clause. Next, the specified rows are grouped according to the GROUP BY clause. Then, the group is specified by the search conditions in the HAVING clause.

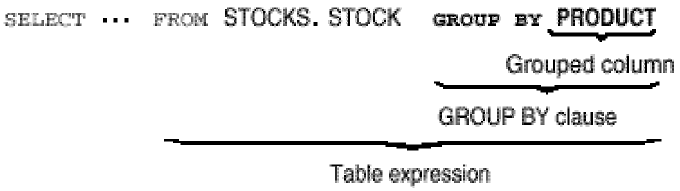
6.1.1 Grouping tables and fetching data

■Grouping tables

The first method of grouping tables is as follows:

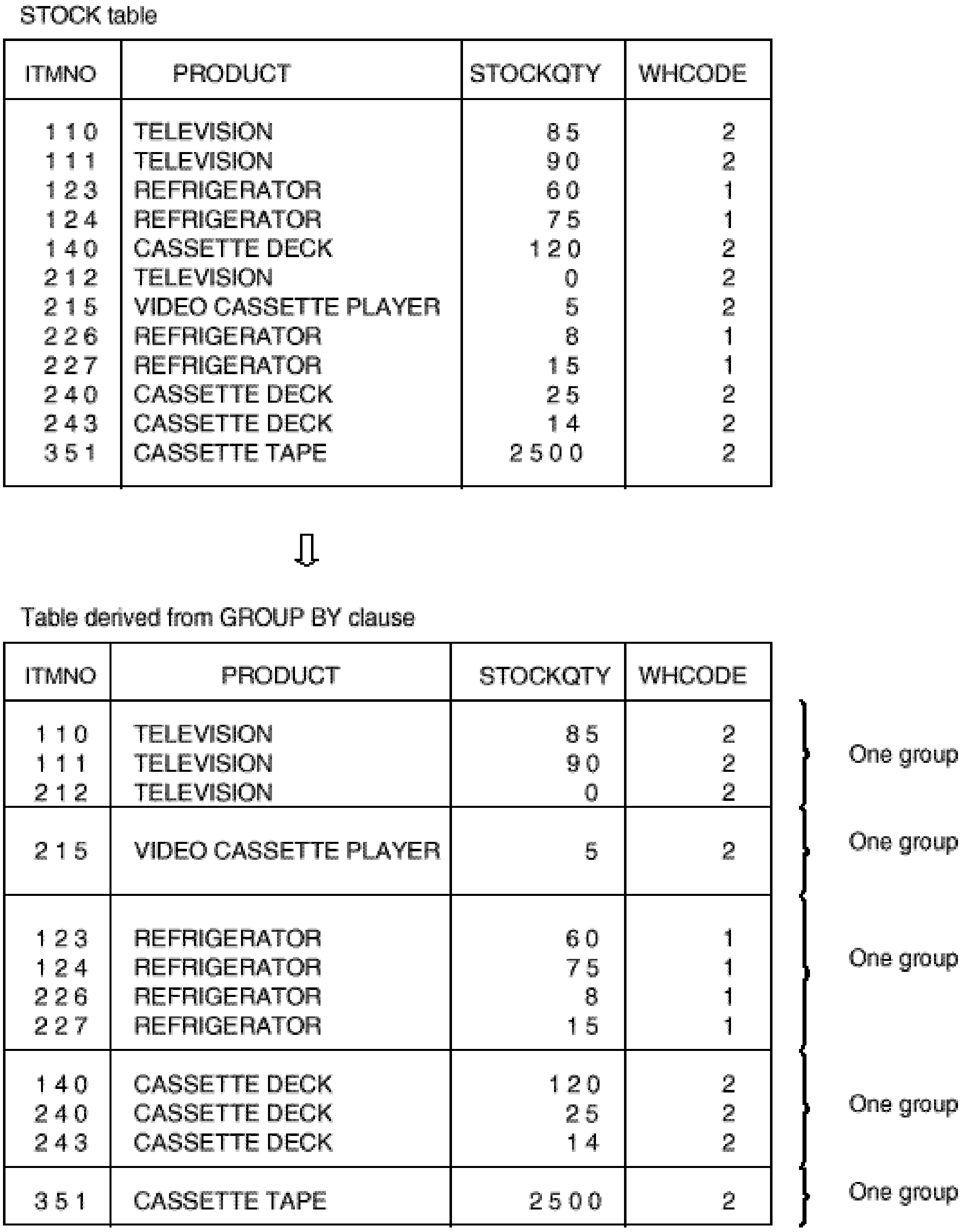
Example 1:

In this example, the STOCK table is grouped according to the values for PRODUCT.



"Figure: Example of a table derived from GROUP BY clause" shows the table derived from the GROUP BY clause.

[Figure: Example of a table derived from GROUP BY clause]



STOCK table

ITMNO	PRODUCT	STOCKQTY	WHCODE
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
140	CASSETTE DECK	120	2
212	TELEVISION	0	2
215	VIDEO CASSETTE PLAYER	5	2
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETTE TAPE	2500	2

Table derived from GROUP BY clause

ITMNO	PRODUCT	STOCKQTY	WHCODE
110 111 212	TELEVISION TELEVISION TELEVISION	85 90 0	2 2 2
215	VIDEO CASSETTE PLAYER	5	2
123 124 226 227	REFRIGERATOR REFRIGERATOR REFRIGERATOR REFRIGERATOR	60 75 8 15	1 1 1 1
140 240 243	CASSETTE DECK CASSETTE DECK CASSETTE DECK	120 25 14	2 2 2
351	CASSETTE TAPE	2500	2

A collection of rows derived from the FROM clause and WHERE clause in the table expression are grouped together. In this example, the result of the FROM clause is equivalent to the STOCK table because only the STOCK table is specified in the FROM clause in the table expression. Also, because the WHERE clause is omitted, all rows are grouped.

◆GROUP BY clause

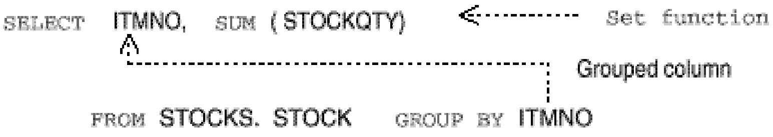
Specify a column name as a group key, a data column value function, or a numeric function in the GROUP BY clause. When a column name is specified, the column is called a grouped column. When a data column value function or numeric function is specified, the function is called a grouped function. In Example 1, the grouped column is PRODUCT. The table derived from the results of the GROUP BY clause is a grouped table in which rows having the same value for the grouped column are formed into one group. When specifying multiple columns as grouped columns, separate the column names with commas (,). In this case, the rows having the same values for each grouped column are formed into one group.

■Fetching data from a grouped table

A query specification is used to derive a table with which data is to be manipulated. If a query specification is used to derive a table from a grouped table, the table is made up of a row for each group. In a query specification for which a GROUP BY clause is specified, a grouped column, a grouped function, or a set function can be specified in the select column list. If a set function is specified, the total, average, maximum, and minimum values, and the row count for the column can be obtained for each group. If a column other than a grouped column is specified by other than a set function, an error occurs. For example, suppose STOCKQTY is specified in the select column list in the query specification in Example 1. In this case, whether the value for STOCKQTY for PRODUCT TELEVISION is "85", "90", or "0" cannot be determined. However, a total value and an average value for these values can be determined, so a set function can be specified. Also, in Figure: Example of a table derived from GROUP BY clause, the values of WHCODE cannot be directly specified in the select column list despite the values being unique in each group. This is the case because a value that is not unique may be produced at any time due to variations in the data. If columns PRODUCT and WHCODE are specified as grouped columns for the data in Figure: Example of a table derived from GROUP BY clause, the values of WHCODE values can be fetched. A literal or host variable can be specified in the select column list

Example 2:

In this example, the totals for PRODUCT and STOCKQTY are obtained from the grouped table in Example 1.



```
SELECT ITMNO, SUM (STOCKQTY)
  FROM STOCKS. STOCK   GROUP BY ITMNO
```

[Figure: Example of fetching data by grouping a table]

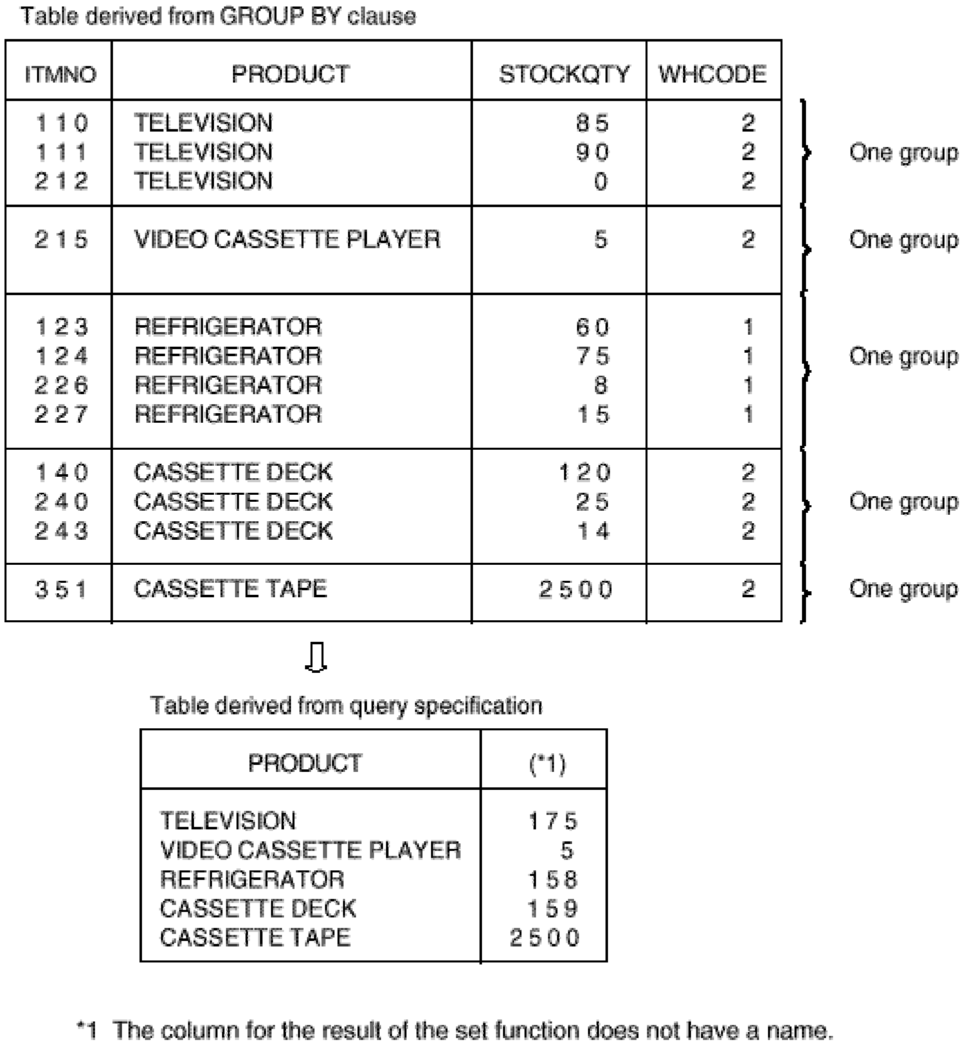


Table derived from GROUP BY clause

ITMNO	PRODUCT	STOCKQTY	WHCODE
110	TELEVISION	85	2
111	TELEVISION	90	2
212	TELEVISION	0	2
215	VIDEO CASSETTE PLAYER	5	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
140	CASSETTE DECK	120	2
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETTE TAPE	2500	2

One group
One group
One group
One group
One group
⇩

Table derived from query specification

PRODUCT	(*1)
TELEVISION	175
VIDEO CASSETTE PLAYER	5
REFRIGERATOR	158
CASSETTE DECK	159
CASSETTE TAPE	2500

*1 The column for the result of the set function does not have a name.

The reader probably already noticed in Figure: Example of a table derived from GROUP BY clause that the result of specifying only a grouped column in the select column list is the same as specifying "DISTINCT" in the query specification. In Example 2, only PRODUCT was specified in the select column list, so PRODUCT was obtained with duplicate rows deleted. However, if the objective is to delete duplicate rows, specify DISTINCT rather than specifying the GROUP BY clause to improve the processing efficiency of the SQL statements.

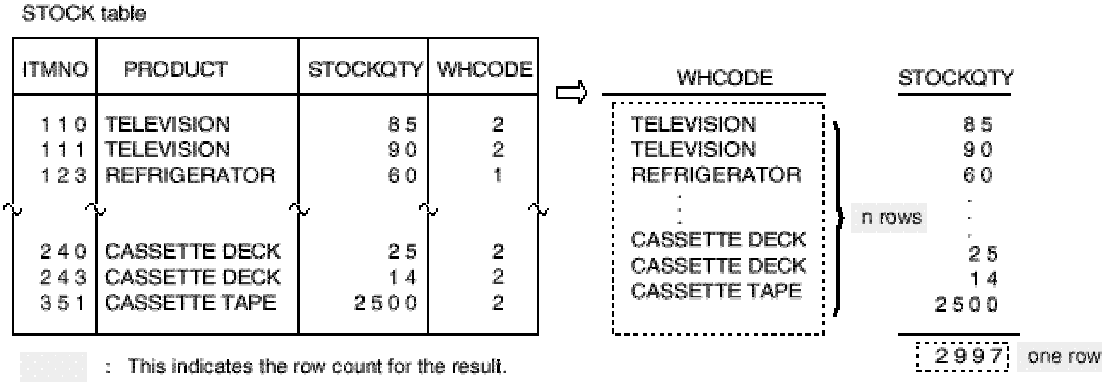
To fetch the result of a set function and a column name as in Example 2, a table must be grouped. If the set function and column name are specified in the select column list but the GROUP BY clause is not specified, an error occurs. The following is an example in which an error occurs:

Example 3:

In this example, a set function is incorrectly used.

```
SELECT  PRODUCT,  SUM (STOCKQTY) FROM  STOCKS. STOCK
```

[Figure: Example of using a set function incorrectly]


STOCK table
ITMNO
PRODUCT
STOCKQTY
WHCODE
110 TELEVISION 85 2
111 TELEVISION 90 2
123 REFRIGERATOR 60 1
240 CASSETTE DECK 25 2
243 CASSETTE DECK 14 2
351 CASSETTE TAPE 2500 2
WHCODE
STOCKQTY
TELEVISION
TELEVISION
REFRIGERATOR
CASSETTE DECK
CASSETTE DECK
CASSETTE TAPE
n rows
85
90
60
25
14
2500
2997
one row
: This indicates the row count for the result.

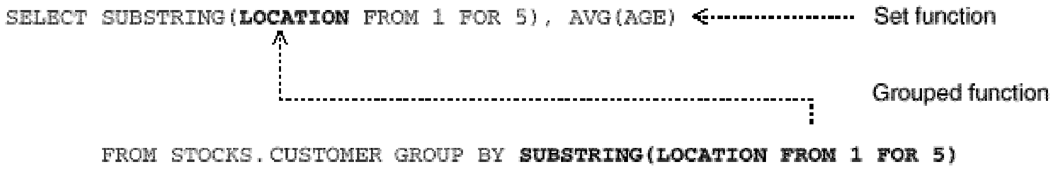
In Example 3, the row count for the result is one row for the total of STOCKQTY, but n rows for PRODUCT. Therefore, this query specification results in an error.

■Grouping tables by using parts of character strings

Methods of grouping table data in columns have been explained so far. The following explains a method of grouping tables by using parts of character strings. Specify a character substring function in the GROUP BY clause. Figure: Example of grouping tables by using parts of character strings shows an example of the specification. For details on the character substring function, see 6.3 "Manipulating Numeric Data".

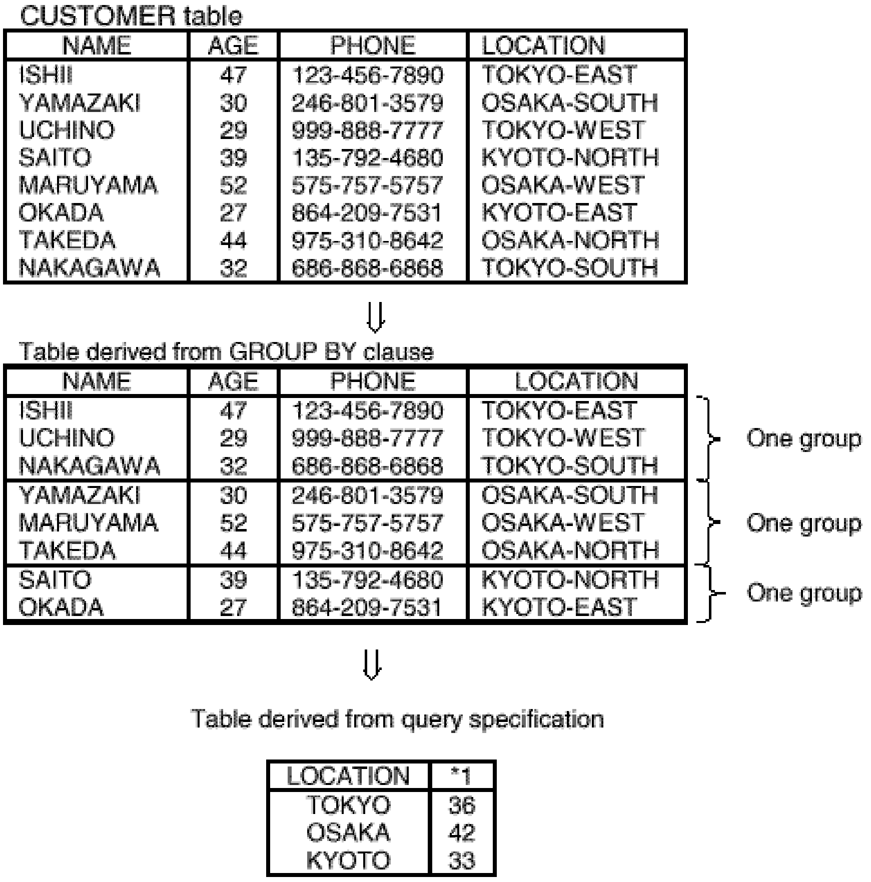
Example 4:

In this example, the city names of column "LOCATION" in the CUSTOMER table are grouped with the same persons in order to fetch the average age of each location.


Set function
Grouped function

```
SELECT SUBSTRING(LOCATION FROM 1 FOR 5), AVG(AGE)
    FROM STOCKS.CUSTOMER GROUP BY SUBSTRING(LOCATION FROM 1 FOR 5)
```

[Figure: Example of grouping tables by using parts of character strings]



CUSTOMER table

NAME	AGE	PHONE	LOCATION
ISHII	47	123-456-7890	TOKYO-EAST
YAMAZAKI	30	246-801-3579	OSAKA-SOUTH
UCHINO	29	999-888-7777	TOKYO-WEST
SAITO	39	135-792-4680	KYOTO-NORTH
MARUYAMA	52	575-757-5757	OSAKA-WEST
OKADA	27	864-209-7531	KYOTO-EAST
TAKEDA	44	975-310-8642	OSAKA-NORTH
NAKAGAWA	32	686-868-6868	TOKYO-SOUTH

Table derived from GROUP BY clause

NAME	AGE	PHONE	LOCATION
ISHII	47	123-456-7890	TOKYO-EAST
UCHINO	29	999-888-7777	TOKYO-WEST
NAKAGAWA	32	686-868-6868	TOKYO-SOUTH
YAMAZAKI	30	246-801-3579	OSAKA-SOUTH
MARUYAMA	52	575-757-5757	OSAKA-WEST
TAKEDA	44	975-310-8642	OSAKA-NORTH
SAITO	39	135-792-4680	KYOTO-NORTH
OKADA	27	864-209-7531	KYOTO-EAST

Table derived from query specification

LOCATION	*1
TOKYO	36
OSAKA	42
KYOTO	33

*1 The column for the results of the set function does not have a name.

■Case-splitting data and grouping tables

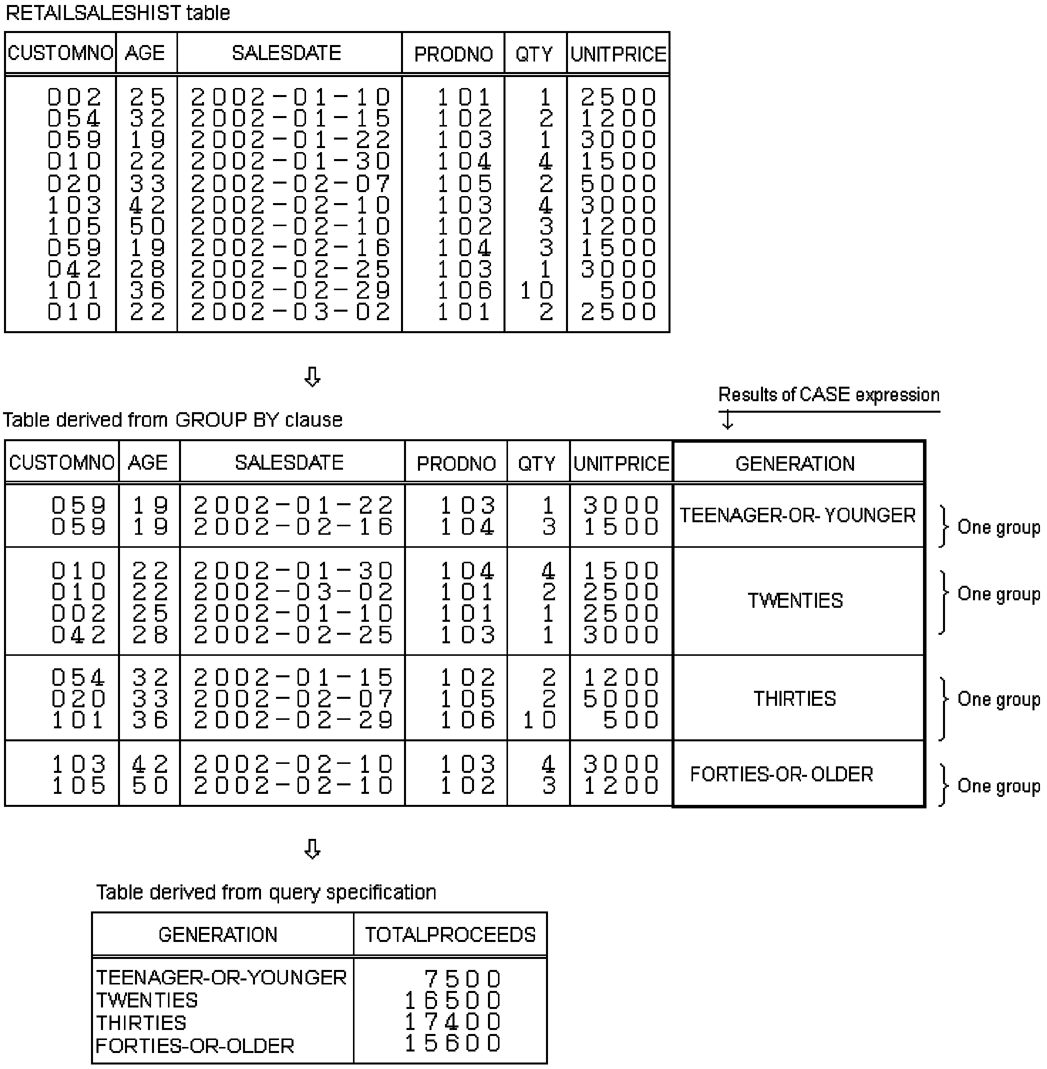
Specify a CASE expression in the GROUP BY clause to split column data into cases instead of columns, and to group tables for each value split into cases. An example of specification is shown below. For details on the CASE expression, refer to "6.7 Using Case Expression to Manipulate Data."

Example 5

The retail-sale history table is grouped for each generation, and the total proceeds are fetched for each generation.

```
SELECT GENERATION, SUM(UNITPRICE*QTY) AS TOTALPROCEEDS FROM  RETAILSALESHIS
      GROUP BY CASE WHEN AGE < 20 THEN 'TEENAGER-OR-YOUNGER'
                    WHEN AGE < 30 THEN 'TWENTIES'
                    WHEN AGE < 40 THEN 'THIRTIES'
                                  ELSE 'FORTIES-OR-OLDER' END AS GENERATION
```

[Figure: Case-splitting data and grouping tables]



RETAILSALESHIST table

CUSTOMNO	AGE	SALESDATE	PRODNO	QTY	UNITPRICE
002	25	2002-01-10	101	1	2500
054	32	2002-01-15	102	2	1200
059	19	2002-01-22	103	1	3000
010	22	2002-01-30	104	4	1500
020	33	2002-02-07	105	2	5000
103	42	2002-02-10	103	4	3000
105	50	2002-02-10	102	3	1200
059	19	2002-02-16	104	3	1500
042	28	2002-02-25	103	1	3000
101	36	2002-02-29	106	10	500
010	22	2002-03-02	101	2	2500

Table derived from GROUP BY clause

CUSTOMNO	AGE	SALESDATE	PRODNO	QTY	UNITPRICE	GENERATION
059	19	2002-01-22	103	1	3000	TEENAGER-OR- YOUNGER
059	19	2002-02-16	104	3	1500	
010	22	2002-01-30	104	4	1500	TWENTIES
010	22	2002-03-02	101	2	2500	
002	25	2002-01-10	101	1	2500	
042	28	2002-02-25	103	1	3000	
054	32	2002-01-15	102	2	1200	THIRTIES
020	33	2002-02-07	105	2	5000	
101	36	2002-02-29	106	10	500	
103	42	2002-02-10	103	4	3000	FORTIES-OR- OLDER
105	50	2002-02-10	102	3	1200	

Table derived from query specification

GENERATION	TOTALPROCEEDS
TEENAGER-OR-YOUNGER	7500
TWENTIES	16500
THIRTIES	17400
FORTIES-OR-OLDER	15600

■Grouping tables by month

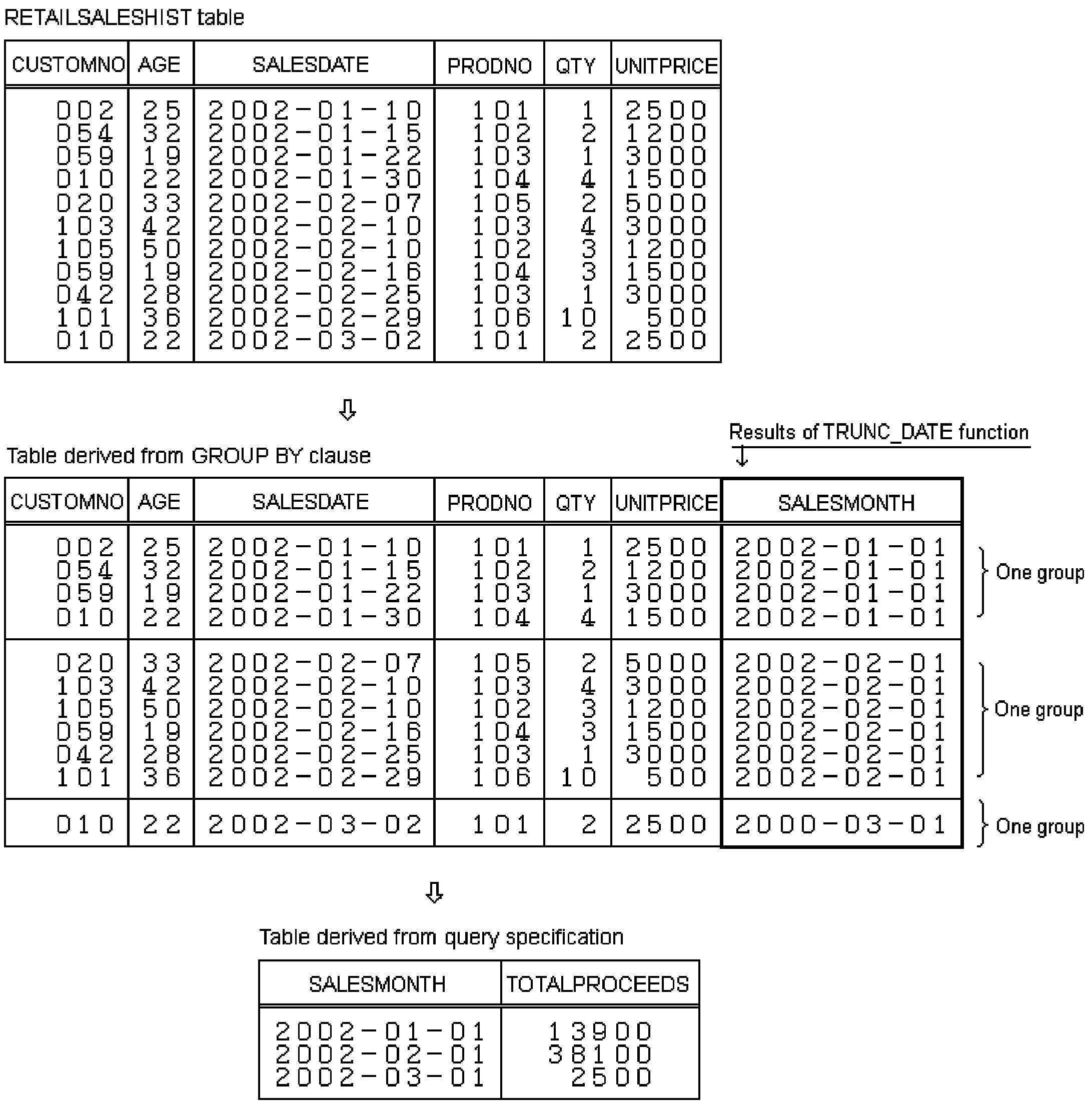
To group tables of a DATE-type column by year or month, specify the date-time value function. A specification example is shown below. For details on the date-time value function, see "6.5 Manipulating Date Data."

Example 6

The sales dates in the RETAILSALESHIST table are grouped by month, and the total proceeds are fetched.

```
SELECT SALESMONTH, SUM(UNITPRICE*QTY) AS SALES PROCEEDS FROM RETAILSALESHIST
   GROUP BY TRUNC_DATE(SALESDATE,'MONTH') AS SALESMONTH
```

[Figure: Example of grouping tables by month]



RETAILSALESHIST table

CUSTOMNO	AGE	SALESDATE	PRODNO	QTY	UNITPRICE
002	25	2002-01-10	101	1	2500
054	32	2002-01-15	102	2	1200
059	19	2002-01-22	103	1	3000
010	22	2002-01-30	104	4	1500
020	33	2002-02-07	105	2	5000
103	42	2002-02-10	103	4	3000
105	50	2002-02-10	102	3	1200
059	19	2002-02-16	104	3	1500
042	28	2002-02-25	103	1	3000
101	36	2002-02-29	106	10	500
010	22	2002-03-02	101	2	2500

⇩

Table derived from GROUP BY clause

Results of TRUNC_DATE function

CUSTOMNO	AGE	SALESDATE	PRODNO	QTY	UNITPRICE	SALESMONTH
002	25	2002-01-10	101	1	2500	2002-01-01
054	32	2002-01-15	102	2	1200	2002-01-01
059	19	2002-01-22	103	1	3000	2002-01-01
010	22	2002-01-30	104	4	1500	2002-01-01
020	33	2002-02-07	105	2	5000	2002-02-01
103	42	2002-02-10	103	4	3000	2002-02-01
105	50	2002-02-10	102	3	1200	2002-02-01
059	19	2002-02-16	104	3	1500	2002-02-01
042	28	2002-02-25	103	1	3000	2002-02-01
101	36	2002-02-29	106	10	500	2002-02-01
010	22	2002-03-02	101	2	2500	2000-03-01

⇩

Table derived from query specification

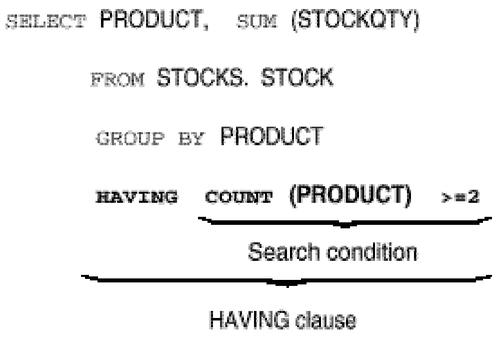
SALESMONTH	TOTALPROCEEDS
2002-01-01	13900
2002-02-01	38100
2002-03-01	2500

6.1.2 Specifying groups to be processed from a grouped table

The WHERE clause can be used in a table expression to specify rows to be processed in the table specified in the FROM clause. To do this, specifying the groups to be processed in a table grouped by the GROUP BY clause, using a HAVING clause. The following is an example of specifying the HAVING clause:

Example:

In this example, the totals of ITMNO and STOCKQTY are obtained for products with two or more occurrences of the same ITMNO from the grouped table in Figure: Example of a table derived from GROUP BY clause shows the table derived from the GROUP BY clause.



```
SELECT PRODUCT,  SUM (STOCKQTY)
     FROM STOCKS. STOCK
     GROUP BY PRODUCT
     HAVING  COUNT (PRODUCT)  >=2
```

Search condition
HAVING clause

Figure: Example of a table derived from a HAVING clause shows the table derived from the HAVING clause and the query specification result.

[Figure: Example of a table derived from a HAVING clause]

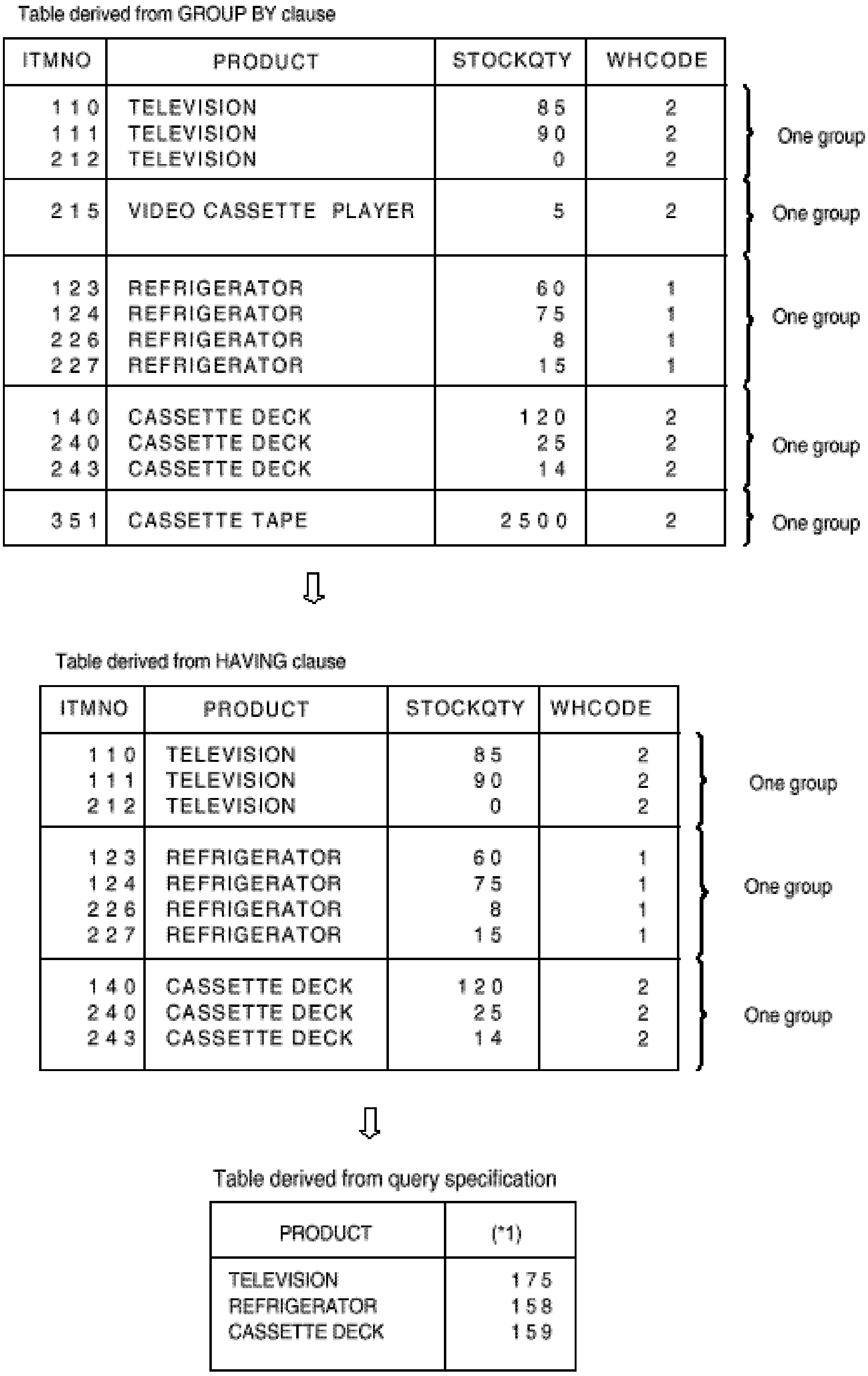


Table derived from GROUP BY clause

ITMNO	PRODUCT	STOCKQTY	WHCODE
110	TELEVISION	85	2
111	TELEVISION	90	2
212	TELEVISION	0	2
215	VIDEO CASSETTE PLAYER	5	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
140	CASSETTE DECK	120	2
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETTE TAPE	2500	2

One group
One group
One group
One group
One group

Table derived from HAVING clause

ITMNO	PRODUCT	STOCKQTY	WHCODE
110	TELEVISION	85	2
111	TELEVISION	90	2
212	TELEVISION	0	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
140	CASSETTE DECK	120	2
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2

Table derived from query specification

PRODUCT	(*1)
TELEVISION	175
REFRIGERATOR	158
CASSETTE DECK	159

*1 The column for the result of the set function does not have a name.

■HAVING clause

Specify the search conditions for the groups to be processed in the HAVING clause. The columns used as conditions must either be grouped columns or must be able to be specified by a set function. If a column other than a grouped column is specified by other than a set function, an error occurs.

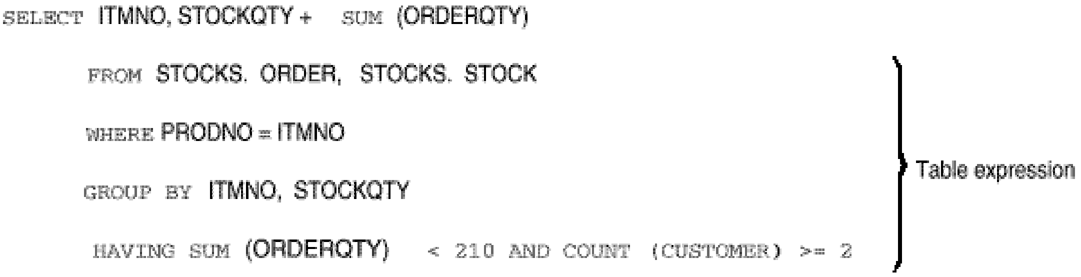
The WHERE clause can also specify search conditions for grouped columns. Whereas the HAVING clause targets grouped tables, the WHERE clause specifies the rows to be processed in the step before being grouped. Therefore, specifying search conditions in the WHERE clause improves the processing efficiency of SQL statements. For example, to derive rows in which PRODUCT is TELEVISION, specify PRODUCT = 'TELEVISION' as the condition in the WHERE clause of the query specification in the previous example.

6.1.3 Grouping multiple tables that are joined and manipulating data

Specifying multiple tables in the FROM clause to join tables and manipulate data has already been explained. A table can be derived from the result of the FROM clause in the table expression. Another table can be derived from the result of a WHERE clause that specifies rows to be processed from the previously derived table. The GROUP BY clause targets this table for processing. The results of joining multiple tables can be grouped and data manipulated. The following is an example of specifying these clauses:

Example 1:

In this example, ITMNO and STOCKQTY for products for which the total of ORDER QTY is less than 210 and for which orders exist for more than one CUSTOMER are added. Then, the total values for ITMNO and STOCKQTY from the STOCK table and the ORDER table are obtained.



```
SELECT ITMNO, STOCKQTY +   SUM (ORDERQTY)
    FROM STOCKS. ORDER,  STOCKS. STOCK
    WHERE PRODNO = ITMNO
    GROUP BY  ITMNO, STOCKQTY
    HAVING SUM (ORDERQTY)   < 210 AND COUNT (CUSTOMER) >= 2
```

Table expression

[Figure: Example of grouping multiple tables that are joined and manipulating data]

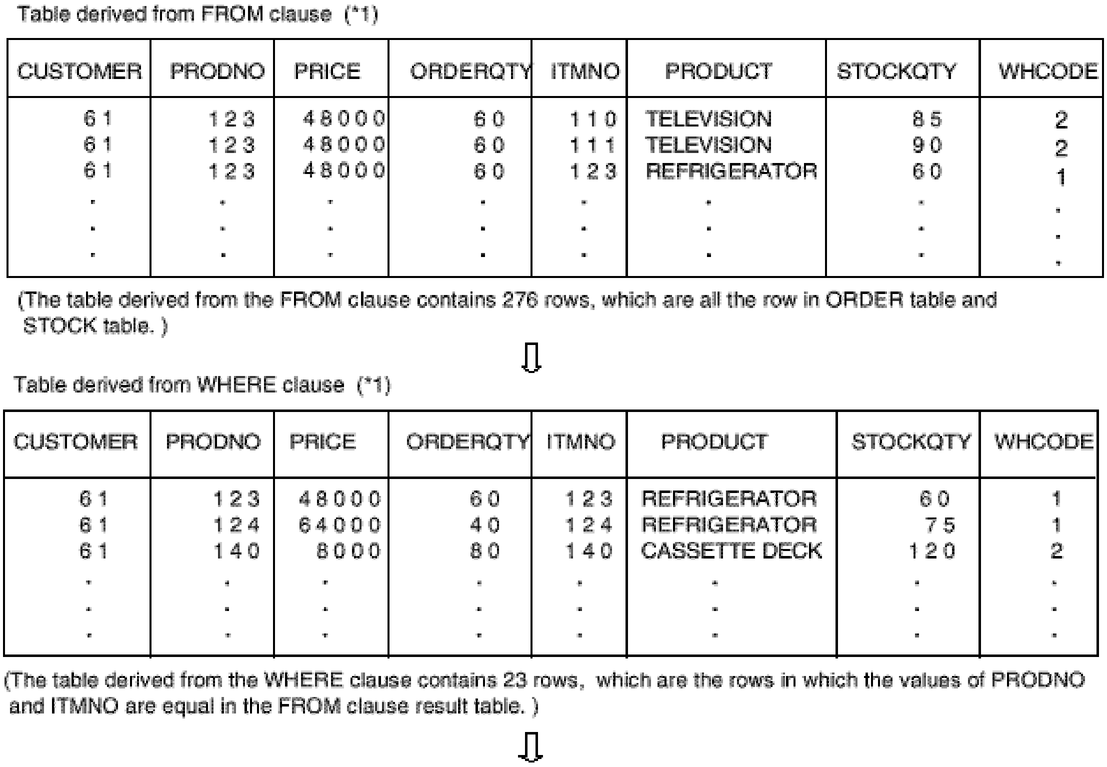


Table derived from FROM clause (*1)

CUSTOMER	PRODNO	PRICE	ORDERQTY	ITMNO	PRODUCT	STOCKQTY	WHCODE
61	123	48000	60	110	TELEVISION	85	2
61	123	48000	60	111	TELEVISION	90	2
61	123	48000	60	123	REFRIGERATOR	60	1
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.

(The table derived from the FROM clause contains 276 rows, which are all the row in ORDER table and STOCK table.)

⇩

Table derived from WHERE clause (*1)

CUSTOMER	PRODNO	PRICE	ORDERQTY	ITMNO	PRODUCT	STOCKQTY	WHCODE
61	123	48000	60	123	REFRIGERATOR	60	1
61	124	64000	40	124	REFRIGERATOR	75	1
61	140	8000	80	140	CASSETTE DECK	120	2
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.

(The table derived from the WHERE clause contains 23 rows, which are the rows in which the values of PRODNO and ITMNO are equal in the FROM clause result table.)

⇩

Table derived from GROUP BY clause

CUSTOMER	PRODNO	PRICE	ORDERQTY	ITMNO	PRODUCT	STOCKQTY	WHCODE
62	110	37500	120	110	TELEVISION	85	2
74	110	39000	120	110	TELEVISION	85	2
63	111	57400	80	111	TELEVISION	90	2
74	111	54000	120	111	TELEVISION	90	2
61	123	48000	60	123	REFRIGERATOR	60	1
61	124	64000	40	124	REFRIGERATOR	75	1
61	140	8000	80	140	CASSETTE DECK	120	2
71	140	7800	50	140	CASSETTE DECK	120	2
72	140	7000	70	140	CASSETTE DECK	120	2
63	212	205000	30	212	TELEVISION	0	2
61	215	240000	10	215	VIDEO CASSETTE PLAYER	5	2
63	215	246000	10	215	VIDEO CASSETTE PLAYER	5	2
72	215	210000	10	215	VIDEO CASSETTE PLAYER	5	2
62	226	112500	20	226	REFRIGERATOR	8	1
72	226	105000	20	226	REFRIGERATOR	8	1
74	226	117000	20	226	REFRIGERATOR	8	1
74	227	140400	10	227	REFRIGERATOR	15	1
61	240	80000	20	240	CASSETTE DECK	25	2
72	243	84000	10	243	CASSETTE DECK	14	2
62	351	375	800	351	CASSETTE TAPE	2500	2
71	351	390	600	351	CASSETTE TAPE	2500	2
72	351	350	1000	351	CASSETTE TAPE	2500	2
74	351	390	700	351	CASSETTE TAPE	2500	2

⇩

Table derived from HAVING clause

CUSTOMER	PRODNO	PRICE	ORDERQTY	ITMNO	PRODUCT	STOCKQTY	WHCODE
63	111	57400	80	111	TELEVISION	90	2
74	111	54000	120	111	TELEVISION	90	2
61	140	8000	80	140	CASSETTE DECK	120	2
71	140	7800	50	140	CASSETTE DECK	120	2
72	140	7000	70	140	CASSETTE DECK	120	2
61	215	240000	10	215	VIDEO CASSETTE PLAYER	5	2
63	215	246000	10	215	VIDEO CASSETTE PLAYER	5	2
72	215	210000	10	215	VIDEO CASSETTE PLAYER	5	2
62	226	112500	20	226	REFRIGERATOR	8	1
72	226	105000	20	226	REFRIGERATOR	8	1
74	226	117000	20	226	REFRIGERATOR	8	1

⇩

Table derived from query specification

ITMNO	(*2)
111	290
140	320
215	35
226	68

*1 See Figure “Example in which multiple tables are joined and data manipulated”, for an explanation of the method of deriving results from the FROM clause and the WHERE clause in this example.

*2 The column for the result of the operational expression does not have a name.

A common error with this method of manipulating data is that the user may mistakenly believe that obtaining the totals of STOCKQTY and ORDERQTY for each ITMNO will provide the same result as in Example 1. The following is an example of using this method incorrectly:

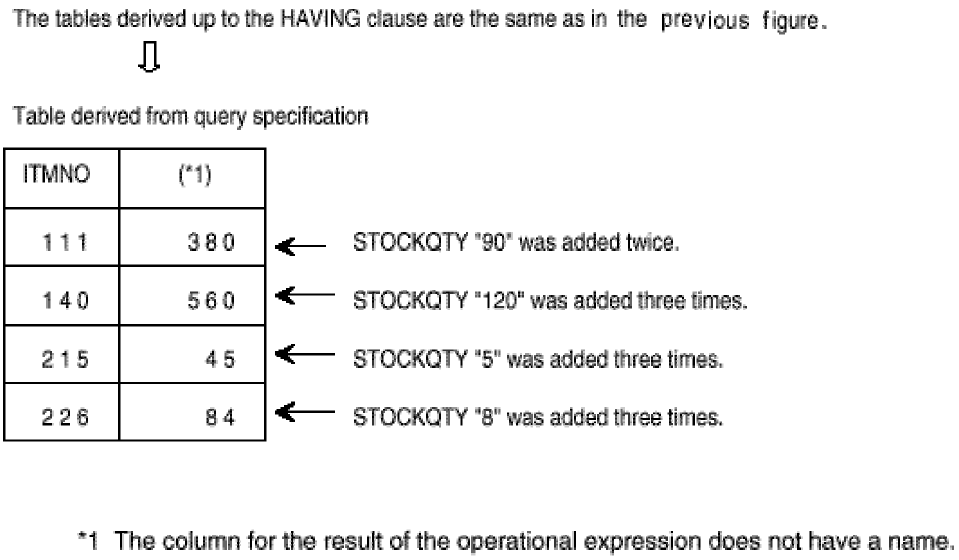
Example 2:

In this example, the data manipulation used in Example 1 is incorrectly specified.

```
SELECT ITMNO, SUM (STOCKQTY) + SUM (ORDERQTY)
            .
            .
            .
   GROUP BY ITMNO
            .
            .
            .
```

Figure: Example of a common error when grouping multiple joined tables shows the result of the execution of this query specification.

[Figure: Example of a common error when grouping multiple joined tables]



The tables derived up to the HAVING clause are the same as in the previous figure.

⇩

Table derived from query specification

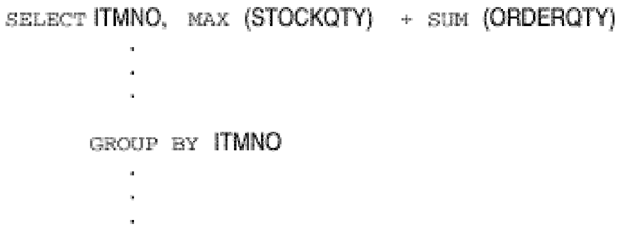
ITMNO	(*1)	
111	380	← STOCKQTY "90" was added twice.
140	560	← STOCKQTY "120" was added three times.
215	45	← STOCKQTY "5" was added three times.
226	84	← STOCKQTY "8" was added three times.

*1 The column for the result of the operational expression does not have a name.

The MAX function or MIN function can be used to correct the previous error, as in the following example:

Example 3:

In this example, the error in Example 2 is corrected.



```
SELECT ITMNO, MAX (STOCKQTY) + SUM (ORDERQTY)
                 .
                 .
                 .
           GROUP BY ITMNO
                 .
                 .
                 .
```

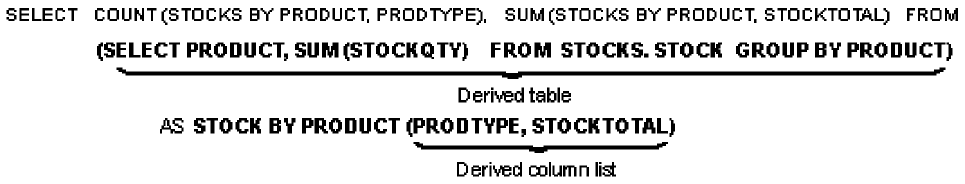
Because the value of STOCKQTY is the same in each group, both the maximum and minimum values for each group can only be one value, the STOCKQTY value. Therefore, the result of this query specification is the same as in Figure: Example of grouping multiple tables that are joined and manipulating data. However, to improve the processing efficiency, the SQL statement should be specified as in Example 1.

6.1.4 Recalculating calculation results for a grouped table

To recalculate results that have already been calculated for a grouped table, use a derived table. When a derived table is used, the results can be recalculated without a view table definition. The following is a specification example:

Example:

Use the derived table used to get the total stock quantity for each product from the stock table, and get the sums for the number of products and stock quantity.



```
SELECT  COUNT(STOCKS BY PRODUCT, PRODTYPE),  SUM(STOCKS BY PRODUCT, STOCKTOTAL)  FROM
        (SELECT PRODUCT, SUM(STOCKQTY)  FROM STOCKS. STOCK  GROUP BY PRODUCT)
                                   Derived table
            AS STOCK BY PRODUCT (PRODTYPE, STOCKTOTAL)
                                   Derived column list
```

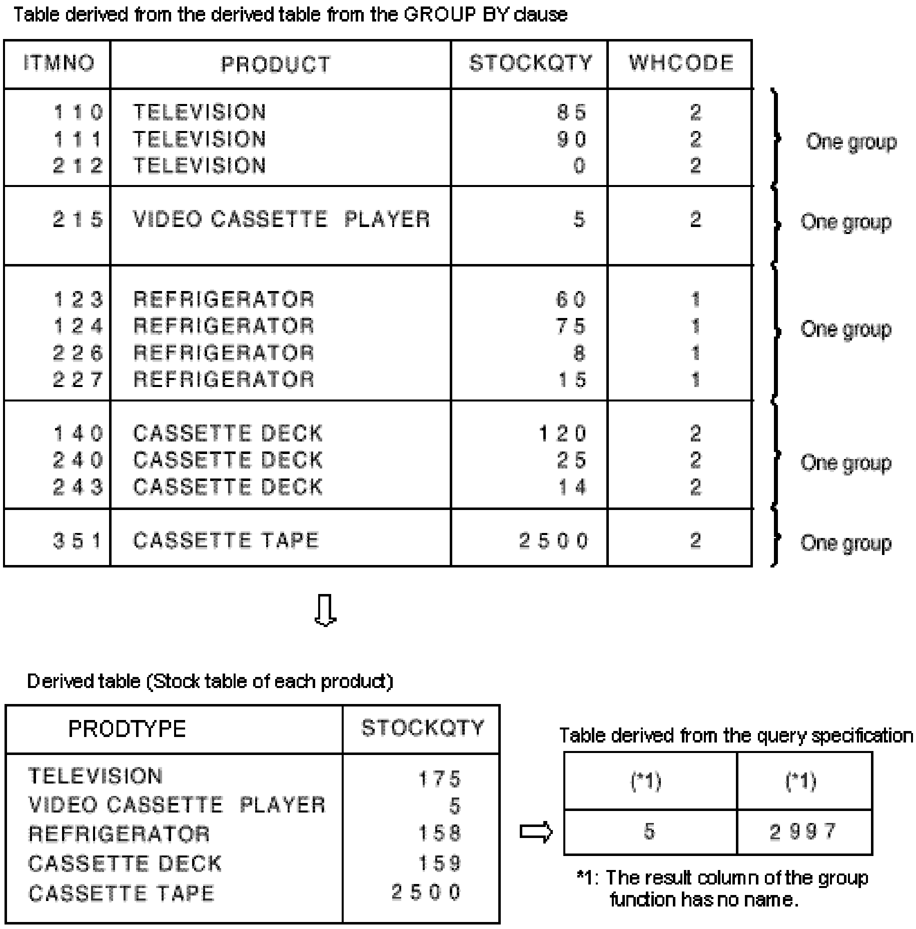


Table derived from the derived table from the GROUP BY clause

ITMNO	PRODUCT	STOCKQTY	WHCODE
110 111 212	TELEVISION TELEVISION TELEVISION	85 90 0	2 2 2
215	VIDEO CASSETTE PLAYER	5	2
123 124 226 227	REFRIGERATOR REFRIGERATOR REFRIGERATOR REFRIGERATOR	60 75 8 15	1 1 1 1
140 240 243	CASSETTE DECK CASSETTE DECK CASSETTE DECK	120 25 14	2 2 2
351	CASSETTE TAPE	2500	2

One group
One group
One group
One group
One group

Derived table (Stock table of each product)

PRODTYPE	STOCKQTY
TELEVISION	175
VIDEO CASSETTE PLAYER	5
REFRIGERATOR	158
CASSETTE DECK	159
CASSETTE TAPE	2500

Table derived from the query specification

(*1)	(*1)
5	2997

*1: The result column of the group function has no name.

6.2 Specifying Various Search Conditions

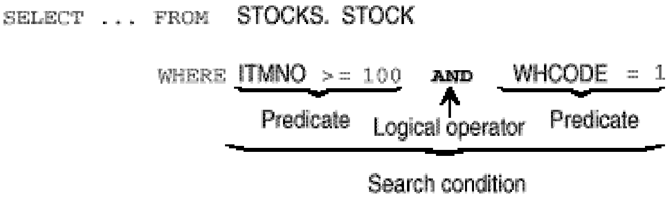
Search conditions can be specified in SQL statements to specify the manipulation of rows that satisfy the search condition. This section explains how to specify search conditions that use a single predicate and predicates combined by logical operators ("AND", "OR", or "NOT"). The result of a predicate or combined predicate is either "true", "false", or "undefined". If the result is "true", the search condition is satisfied. If the result is "false" or "undefined", the search condition is not satisfied.

■Specifying logical operators

This section explains the functions of logical operators and how to specify them.

Example 1:

In this example, the logical operator "AND" is specified in the search condition. The rows from the STOCK table for which ITMNO is 100 or greater and WHCODE is 1 are fetched.

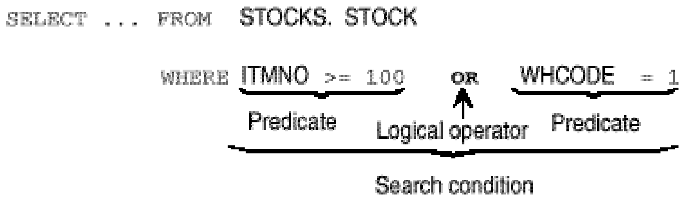


```
SELECT ... FROM STOCKS. STOCK
           WHERE ITMNO >= 100 AND WHCODE = 1
```

Example 2:

In this example, the logical operator "OR" is specified in the search condition. The rows from the

STOCK table for which ITMNO is 100 or greater or WHCODE is 1 are fetched.


SELECT ... FROM STOCKS. STOCK
WHERE ITMNO >= 100 OR WHCODE = 1
Predicate
Logical operator
Predicate
Search condition

Example 3:

In this example, the logical operator "NOT" is specified in the search condition. The rows from the STOCK table for all items except TELEVISION are fetched.

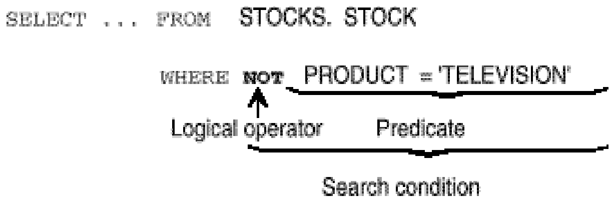

SELECT ... FROM STOCKS. STOCK
WHERE NOT PRODUCT = 'TELEVISION'
Logical operator
Predicate
Search condition

Table: Truth table for AND lists the result of specifying AND between predicates. Table: Truth table for OR lists the result of specifying OR between predicates. Table: Truth table for NOT lists the result of specifying NOT before the predicate.

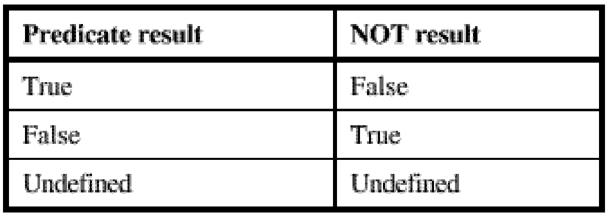
[Table: Truth table for AND]

	Predicate result		
Predicate result	**True**	**False**	**Undefined**
True	True	False	Undefined
False	False	False	False
Undefined	Undefined	False	Undefined

[Table: Truth table for OR]

	Predicate result		
Predicate result	**True**	**False**	**Undefined**
True	True	True	True
False	True	False	Undefined
Undefined	True	Undefined	Undefined

[Table: Truth table for NOT]



Predicate result	NOT result
True	False
False	True
Undefined	Undefined

■Evaluation sequence of logical operators

Multiple logical operators can be specified in a search condition. In this case, logical operators are evaluated in the following sequence:

1. NOT
2. AND
3. OR

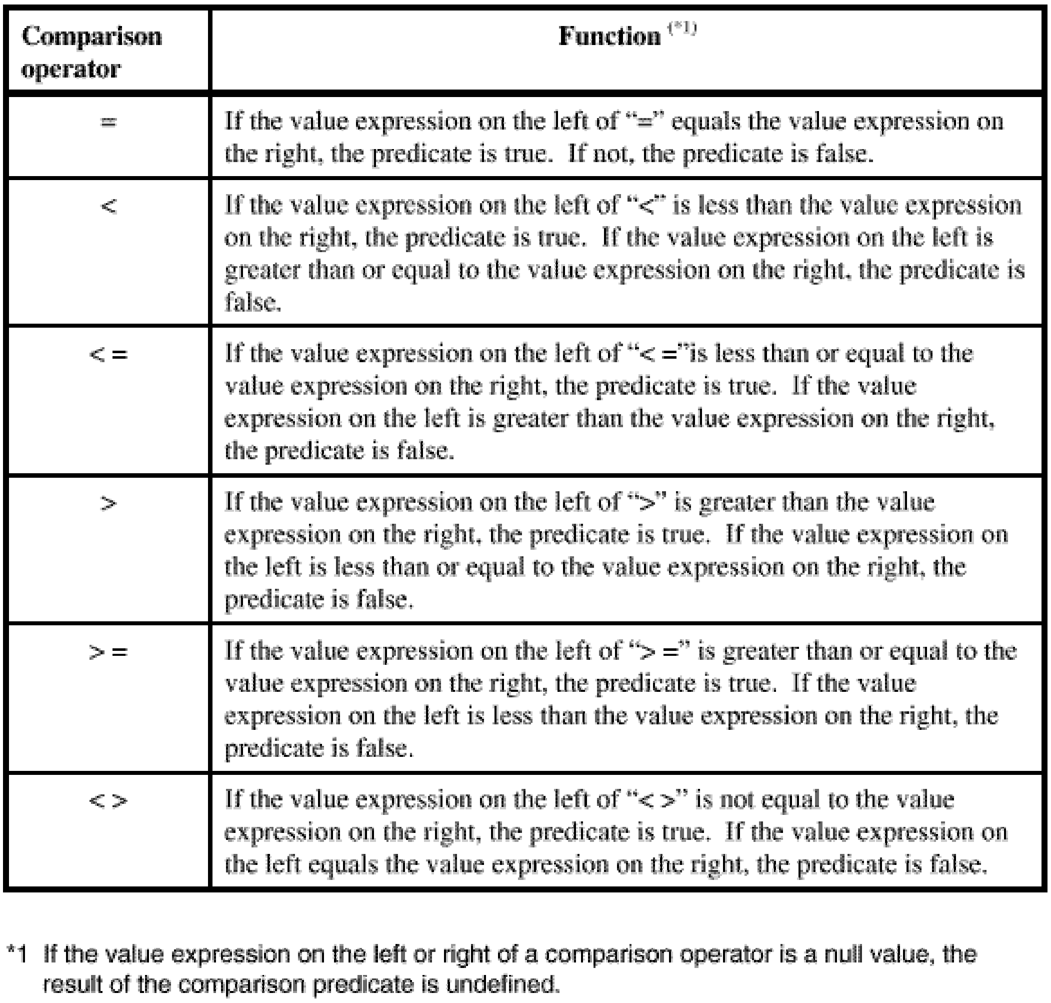
However, parentheses can be used to in the specification of the evaluation sequence. A condition enclosed in parentheses is evaluated before other conditions.

6.2.1 Comparing two values

■Using a comparison operator to compare two values

Use a comparison predicate to compare two values. Table: Comparison operators lists the comparison operators that can be specified in a comparison predicate.

[Table: Comparison operators]

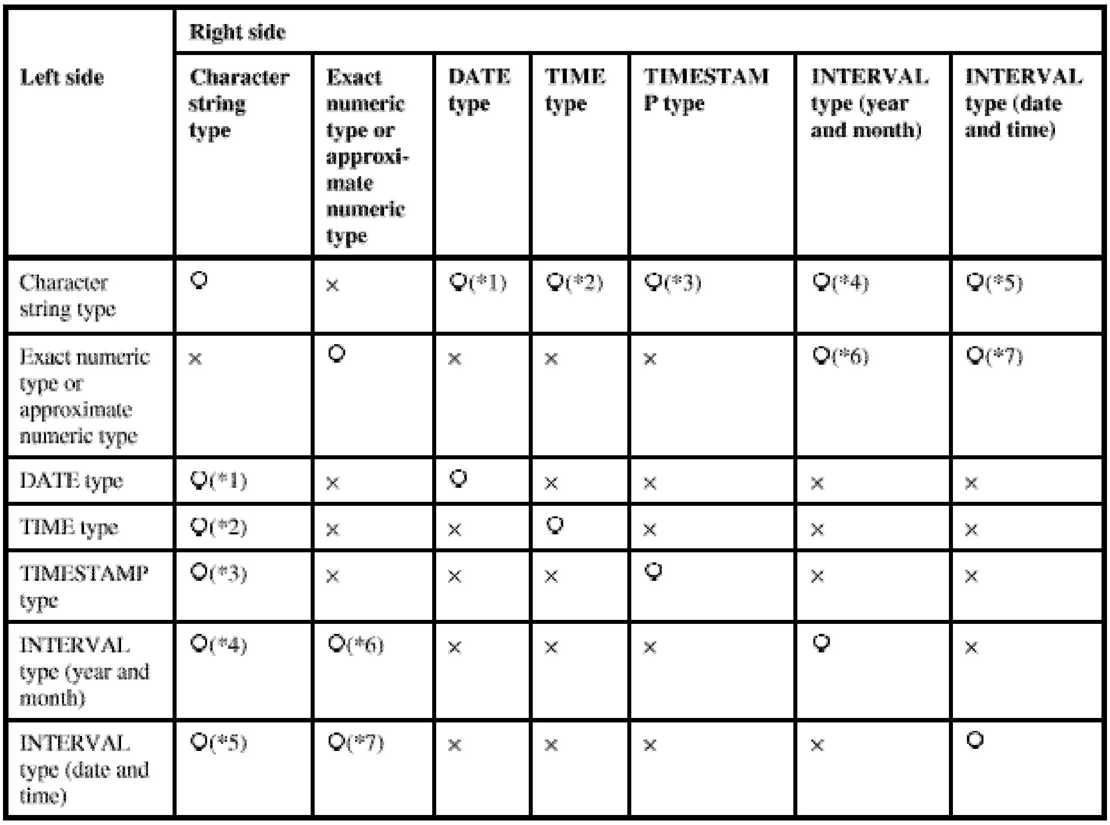


Comparison operator	Function (*1)
=	If the value expression on the left of "=" equals the value expression on the right, the predicate is true. If not, the predicate is false.
<	If the value expression on the left of "<" is less than the value expression on the right, the predicate is true. If the value expression on the left is greater than or equal to the value expression on the right, the predicate is false.
< =	If the value expression on the left of "< ="is less than or equal to the value expression on the right, the predicate is true. If the value expression on the left is greater than the value expression on the right, the predicate is false.
>	If the value expression on the left of ">" is greater than the value expression on the right, the predicate is true. If the value expression on the left is less than or equal to the value expression on the right, the predicate is false.
> =	If the value expression on the left of "> =" is greater than or equal to the value expression on the right, the predicate is true. If the value expression on the left is less than the value expression on the right, the predicate is false.
< >	If the value expression on the left of "< >" is not equal to the value expression on the right, the predicate is true. If the value expression on the left equals the value expression on the right, the predicate is false.

*1 If the value expression on the left or right of a comparison operator is a null value, the result of the comparison predicate is undefined.

The data types of the left and right value expressions of a comparison operator must be comparable. Table: Comparable data types lists comparable data types. For more information on data types, see Appendix A "SQL Data Types and Equivalent Host Variable Data Types".

[Table: Comparable data types]



Left side	Right side						
	Character string type	**Exact numeric type or approximate numeric type**	**DATE type**	**TIME type**	**TIMESTAMP type**	**INTERVAL type (year and month)**	**INTERVAL type (date and time)**
Character string type	○	×	○(*1)	○(*2)	○(*3)	○(*4)	○(*5)
Exact numeric type or approximate numeric type	×	○	×	×	×	○(*6)	○(*7)
DATE type	○(*1)	×	○	×	×	×	×
TIME type	○(*2)	×	×	○	×	×	×
TIMESTAMP type	○(*3)	×	×	×	○	×	×
INTERVAL type (year and month)	○(*4)	○(*6)	×	×	×	○	×
INTERVAL type (date and time)	○(*5)	○(*7)	×	×	×	×	○

○: Can be compared

×: Cannot be compared

*1 Allowed only when the comparison target is a variable specification. The contents of the variable at this time must be a date string.

*2 Allowed only when the comparison target is a variable specification. The contents of the variable at this time must be a time string.

*3 Allowed only when the comparison target is a variable specification. The contents of the variable at this time must be a time-stamp string.

*4 Allowed only when the comparison target is a variable specification. The contents of the variable at this time must be a time interval literal of the year and month class that has the same precision as the comparison source.

*5 Allowed only when the comparison target is a variable specification. The contents of the variable at this time must be a time interval literal of the date and time class that has the same precision as the comparison source.

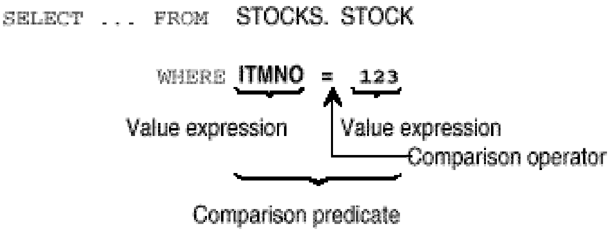
*6 Allowed only when the comparison target is a variable specification of the exact numeric type with decimal scaling position 0. The contents of the variable at this time must be a time interval literal of the year and month class configured from the same single date and time fields as those of the comparison source.

*7 Allowed only when the comparison target is a variable specification of the exact numeric type with decimal scaling position 0. The contents of the variable at this time must be a time interval literal of the date and time class configured from the same single date and time fields as those of the comparison source.

The following is an example of specifying a comparison predicate:

Example 1:

In this example, rows for which ITMNO is 123 are fetched from the STOCK table.


SELECT ... FROM STOCKS. STOCK
WHERE ITMNO = 123
Value expression
Value expression
Comparison operator
Comparison predicate

Example 2:

In this example, rows for which STOCKQTY is less than 50 are fetched from the STOCK table.

```
SELECT ... FROM  STOCKS. STOCK WHERE  STOCKQTY < 50
```

Example 3:

In this example, rows for which "PRICE x ORDERQTY" is 1,000,000 or less are fetched from the ORDER table.

```
SELECT ... FROM  STOCKS. ORDER

     WHERE  PRICE * ORDERQTY  <= 1000000
```

Example 4:

In this example, rows for which STOCKQTY is greater than 500 are fetched from the STOCK table.

```
SELECT ... FROM  STOCKS. STOCK WHERE  STOCKQTY > 500
```

Example 5:

In this example, rows for which STOCKQTY is 500 or greater are fetched from the STOCK table.

```
SELECT ... FROM  STOCKS. STOCK WHERE STOCKQTY >= 500
```

Example 6:

In this example, rows for which PRODUCT is other than TELEVISION are fetched from the STOCK table.

```
SELECT ... FROM  STOCKS. STOCK WHERE  PRODUCT < > 'TELEVISION'
```

The following are examples in which logical operators are used to group and specify several comparison predicates.

Example 7:

In this example, rows for which PRODUCT is TELEVISION and STOCKQTY is 90 or greater are fetched from the STOCK table.

```
SELECT ... FROM  STOCKS. STOCK
```

```
        WHERE PRODUCT = 'TELEVISION' AND STOCKQTY >= 90
```

Example 8:

In this example, rows for which STOCKQTY is 10 or greater and PRODUCT is VIDEO CASSETTE PLAYER, and rows for which STOCKQTY is 200 or greater and PRODUCT is TELEVISION are fetched from the STOCK table.

```
SELECT ... FROM  STOCKS. STOCK

  WHERE STOCKQTY >= 10 AND  PRODUCT = 'VIDEO CASSETTE PLAYER'

    OR STOCKQTY >= 200 AND  PRODUCT = 'TELEVISION'
```

Example 9:

In this example, rows for which PRODUCT is other than TELEVISION or VIDEO CASSETTE PLAYER are fetched from the STOCK table.

```
SELECT ... FROM  STOCKS. STOCK

  WHERE NOT PRODUCT = 'TELEVISION'

    AND NOT PRODUCT = 'VIDEO CASSETTE PLAYER'
```

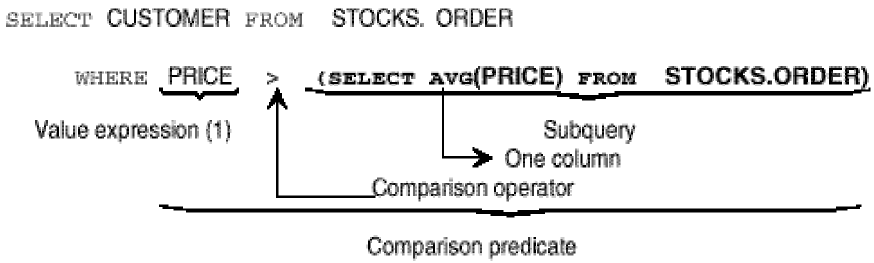
◆Using a subquery in a comparison predicate

If a value specified in a comparison predicate in a search condition is a value derived from a function, a subquery can be coded in the search condition.

The part of the subquery enclosed by parentheses has the same format as the query specification. However, for a subquery, the derived table must have one column. This means that one value expression (column name, function, and so on) must be specified in the left-hand side of the search condition. The following is an example of a comparison predicate in which a subquery is specified:

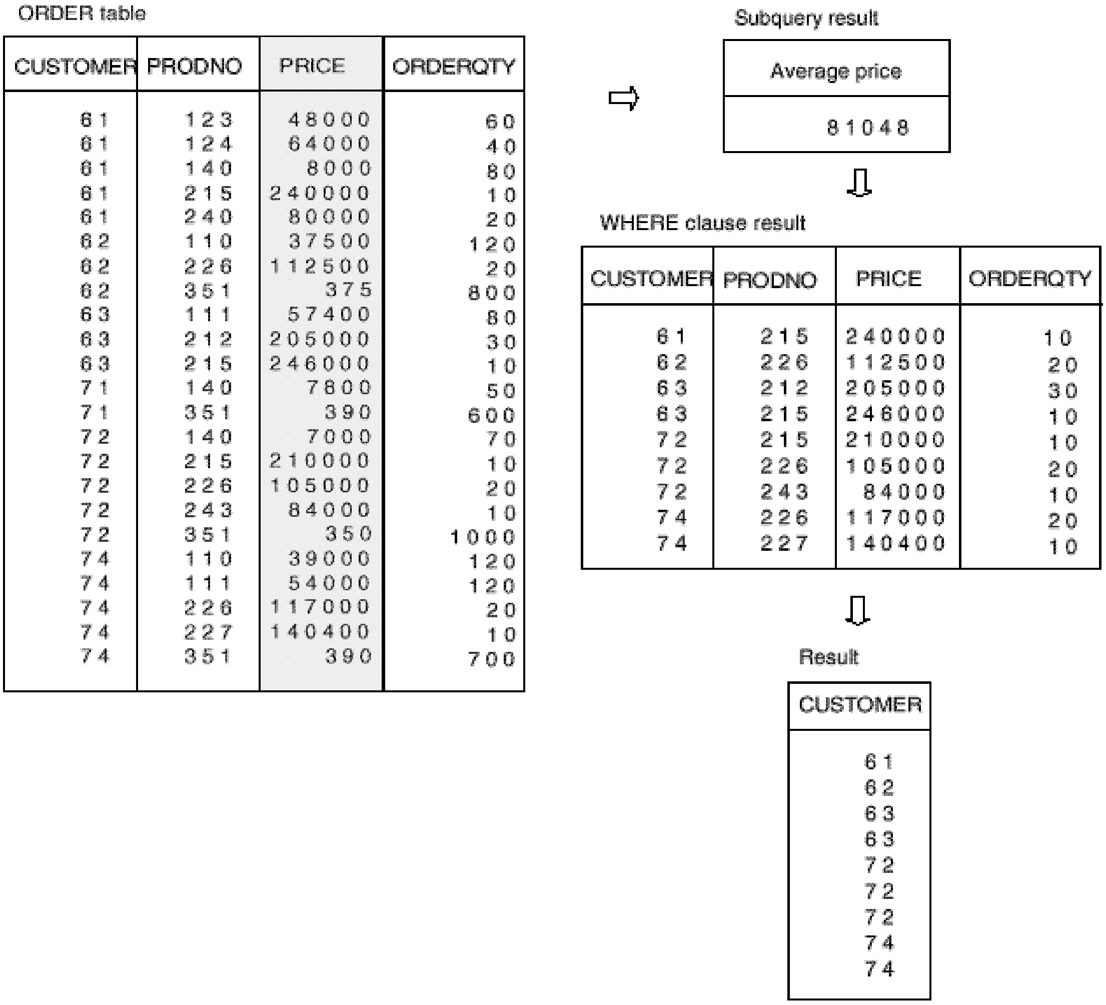
Example 10:

In this example, the average for PRICE is calculated from the ORDER table, and CUSTOMER is fetched from the ORDER table where PRICE is higher than average.



```
SELECT CUSTOMER FROM  STOCKS. ORDER
   WHERE PRICE  >  (SELECT AVG(PRICE) FROM  STOCKS.ORDER)
```

[Figure: Example of specifying a subquery in a comparison predicate to retrieve data]



ORDER table

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	123	48000	60
61	124	64000	40
61	140	8000	80
61	215	240000	10
61	240	80000	20
62	110	37500	120
62	226	112500	20
62	351	375	800
63	111	57400	80
63	212	205000	30
63	215	246000	10
71	140	7800	50
71	351	390	600
72	140	7000	70
72	215	210000	10
72	226	105000	20
72	243	84000	10
72	351	350	1000
74	110	39000	120
74	111	54000	120
74	226	117000	20
74	227	140400	10
74	351	390	700

Subquery result

Average price
81048

WHERE clause result

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	215	240000	10
62	226	112500	20
63	212	205000	30
63	215	246000	10
72	215	210000	10
72	226	105000	20
72	243	84000	10
74	226	117000	20
74	227	140400	10

Result

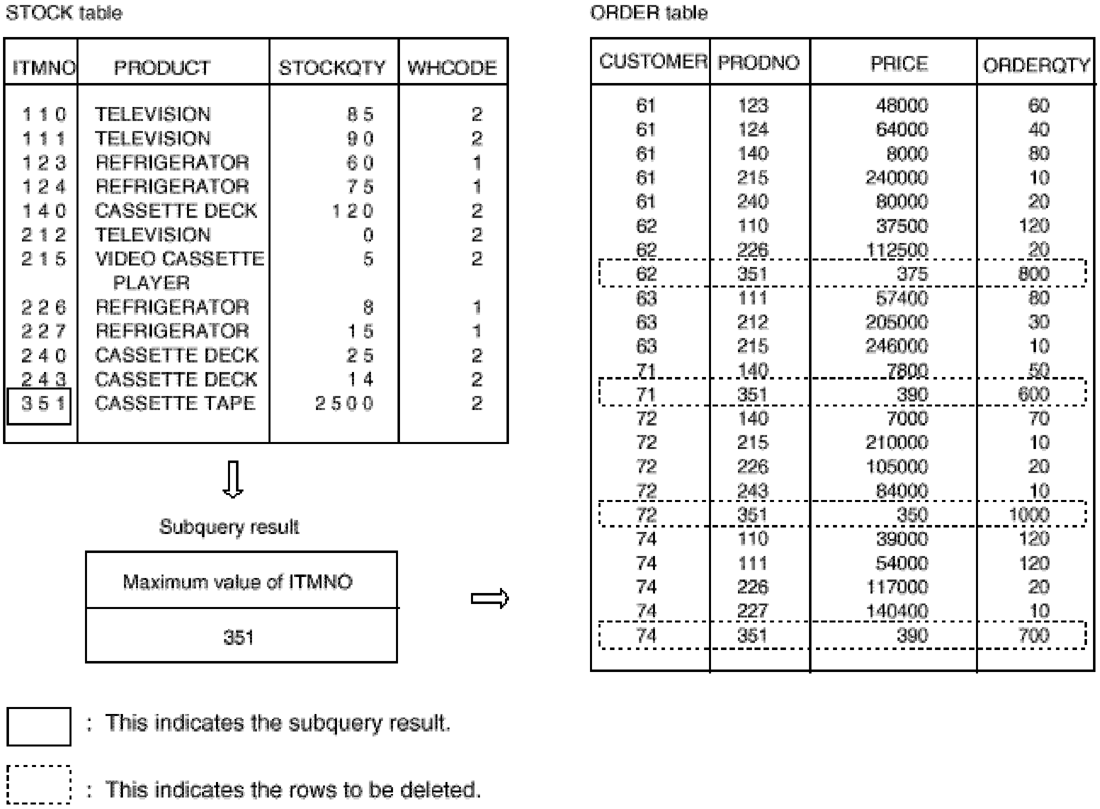
CUSTOMER
61
62
63
63
72
72
72
74
74

Example 11:

In this example, the maximum value for ITMNO is fetched from the STOCK table, and the PRODNO rows for that value are deleted from the ORDER table.

```
DELETE FROM  STOCKS.ORDER

    WHERE  PRODNO = (SELECT MAX (ITMNO) FROM  STOCKS. STOCK)
```

[Figure: Example of specifying a subquery in a comparison predicate to delete data]



STOCK table

ITMNO	PRODUCT	STOCKQTY	WHCODE
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
140	CASSETTE DECK	120	2
212	TELEVISION	0	2
215	VIDEO CASSETTE PLAYER	5	2
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETTE TAPE	2500	2

Subquery result

Maximum value of ITMNO
351

ORDER table

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	123	48000	60
61	124	64000	40
61	140	8000	80
61	215	240000	10
61	240	80000	20
62	110	37500	120
62	226	112500	20
62	351	375	800
63	111	57400	80
63	212	205000	30
63	215	246000	10
71	140	7800	50
71	351	390	600
72	140	7000	70
72	215	210000	10
72	226	105000	20
72	243	84000	10
72	351	350	1000
74	110	39000	120
74	111	54000	120
74	226	117000	20
74	227	140400	10
74	351	390	700

: This indicates the subquery result.

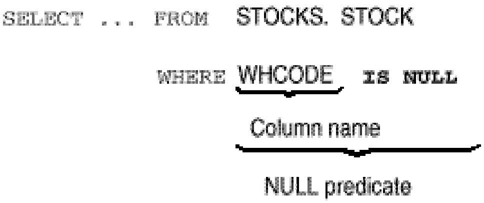
: This indicates the rows to be deleted.

■Checking whether a value is a null value

Use the NULL predicate to check whether a column value is a null value. The following are examples of specifying the NULL predicate:

Example 12:

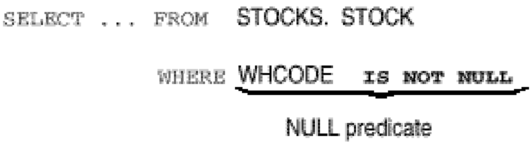
In this example, rows for which WHCODE is a null value are fetched from the STOCK table.



```
SELECT ... FROM STOCKS. STOCK
        WHERE WHCODE IS NULL
```

Example 13:

In this example, rows for which WHCODE is not a null value are fetched from the STOCK table.



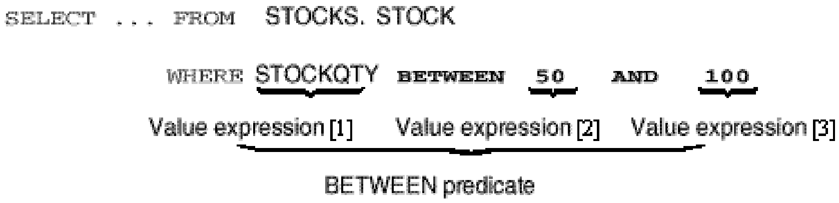
```
SELECT ... FROM STOCKS. STOCK
        WHERE WHCODE IS NOT NULL
```

6.2.2 Checking whether a value is in a certain range

Use the BETWEEN predicate to check whether a value is in a certain range. The following are examples of specifying the BETWEEN predicate:

Example 1:

In this example, rows for which STOCKQTY is greater than or equal to 50 but less than or equal to 100 are fetched from the STOCK table.



```
SELECT ... FROM STOCKS. STOCK
      WHERE STOCKQTY BETWEEN 50 AND 100
```

The result of the BETWEEN predicate is true if the following conditions, based on the three value expressions, are satisfied:

Value expression [2] <= Value expression [1] <= Value expression [3]

The result is false if value expression [1] is less than value expression [2] or greater than value expression [3].

The result is undefined if value expression [1] is a null value or is greater than or equal to value expression [2], and value expression [3] is a null value. The result is also undefined if value expression [1] is less than or equal to value expression [3] and value expression [2] is a null value.

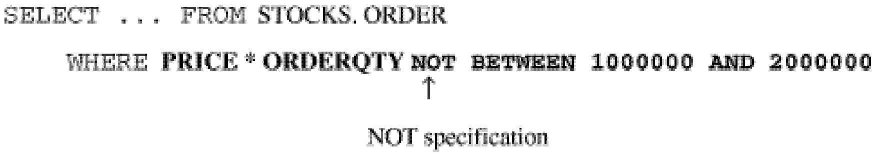
Example 2:

In this example, rows for which STOCKQTY is greater than or equal to 10 but less than or equal to 50, or greater than or equal to 200 but less than or equal to 300 are fetched.

```
SELECT ... FROM STOCKS. STOCK

    WHERE STOCKQTY BETWEEN 10 AND 50

      OR STOCKQTY BETWEEN 200 AND 300
```

Example 3:

In this example, rows for which PRICE x ORDERQTY is less than 1,000,000 or greater than 2,000,000 are fetched from the ORDER table.



```
SELECT ... FROM STOCKS.ORDER
     WHERE PRICE * ORDERQTY NOT BETWEEN 1000000 AND 2000000
```

The result of this specification is the same as in the following:

```
SELECT ... FROM STOCKS.ORDER
     WHERE NOT PRICE * ORDERQTY BETWEEN 1000000 AND 2000000
```

A comparison predicate can also be used to check whether a value is in a certain range. In the following example, the specified comparison predicate has the same effect as the BETWEEN predicate in Example 1:

Example 4:

In this example, rows for which STOCKQTY is greater than or equal to 50 but less than or equal to 100 are fetched from the STOCK table.

```
SELECT ... FROM STOCKS.STOCK WHERE STOCKQTY BETWEEN 50 AND 100
                                   ⇓
SELECT ... FROM STOCKS.STOCK WHERE STOCKQTY >= 50 AND STOCKQTY <=100
```

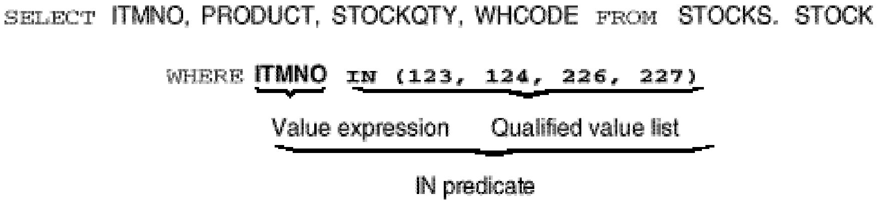
6.2.3 Checking whether values are contained in a set of values

■Checking whether values are contained in a set of values

Use the IN predicate to check whether certain values are contained in a set of quantified values. Use a quantified value list or a subquery to specify the set of values to be compared. The following is an example of specifying a quantified value list:

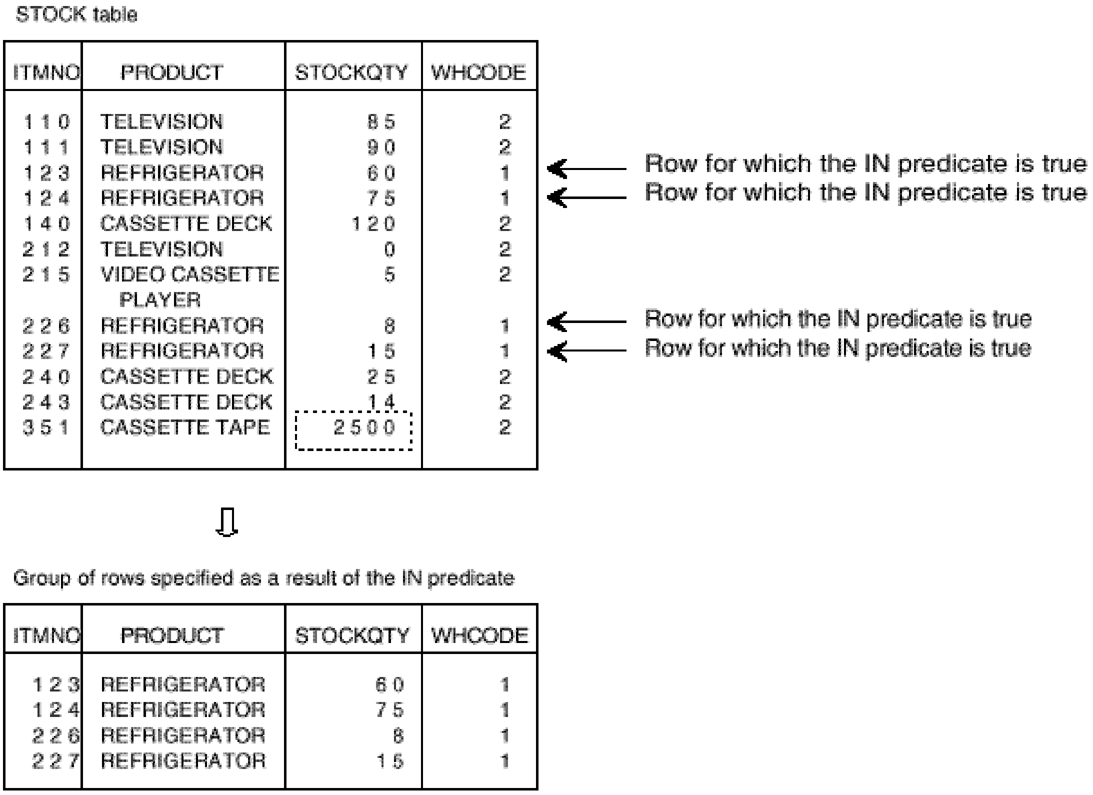
Example 1:

In this example, any row in the STOCK table for which ITMNO is 123, 124, 226, or 227 is specified. Figure: Example of specifying IN predicate shows the result of specifying this quantified value list.



```
SELECT ITMNO, PRODUCT, STOCKQTY, WHCODE FROM STOCKS. STOCK
      WHERE ITMNO IN (123, 124, 226, 227)
```

[Figure: Example of specifying IN predicate]



STOCK table

ITMNO	PRODUCT	STOCKQTY	WHCODE
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
140	CASSETTE DECK	120	2
212	TELEVISION	0	2
215	VIDEO CASSETTE PLAYER	5	2
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETTE TAPE	2500	2

Row for which the IN predicate is true
Row for which the IN predicate is true
Row for which the IN predicate is true
Row for which the IN predicate is true

Group of rows specified as a result of the IN predicate

ITMNO	PRODUCT	STOCKQTY	WHCODE
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1

The values to be compared in the value expression are enclosed in parentheses and specified in a quantified value list. Use host variables or literals to specify values in a quantified value list. Column names cannot be specified.

The result of the IN predicate is true if the value for the value expression is equal to one of the values enclosed in parentheses. The result is false if the value for the value expression does not equal one of the values enclosed in parentheses. The result is undefined if the value of the value expression is a null value or if all the values enclosed in parentheses are null values.

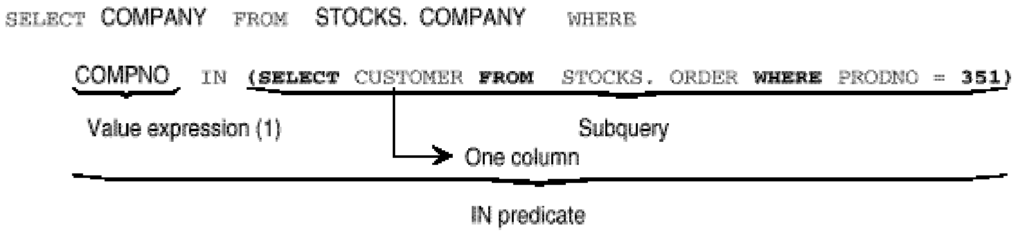
◆Using a subquery in the IN predicate

If the values to be compared to the value expression are values derived from another table, enter a subquery instead of a quantified value list.

The format of the subquery specification in parentheses is the same as that for a query specification. However, the table derived from the subquery must have only one column. This means that one value expression (such as a column name or a function) can be specified in the left-hand side of the search condition. The following is an example of the IN predicate in which a subquery is used:

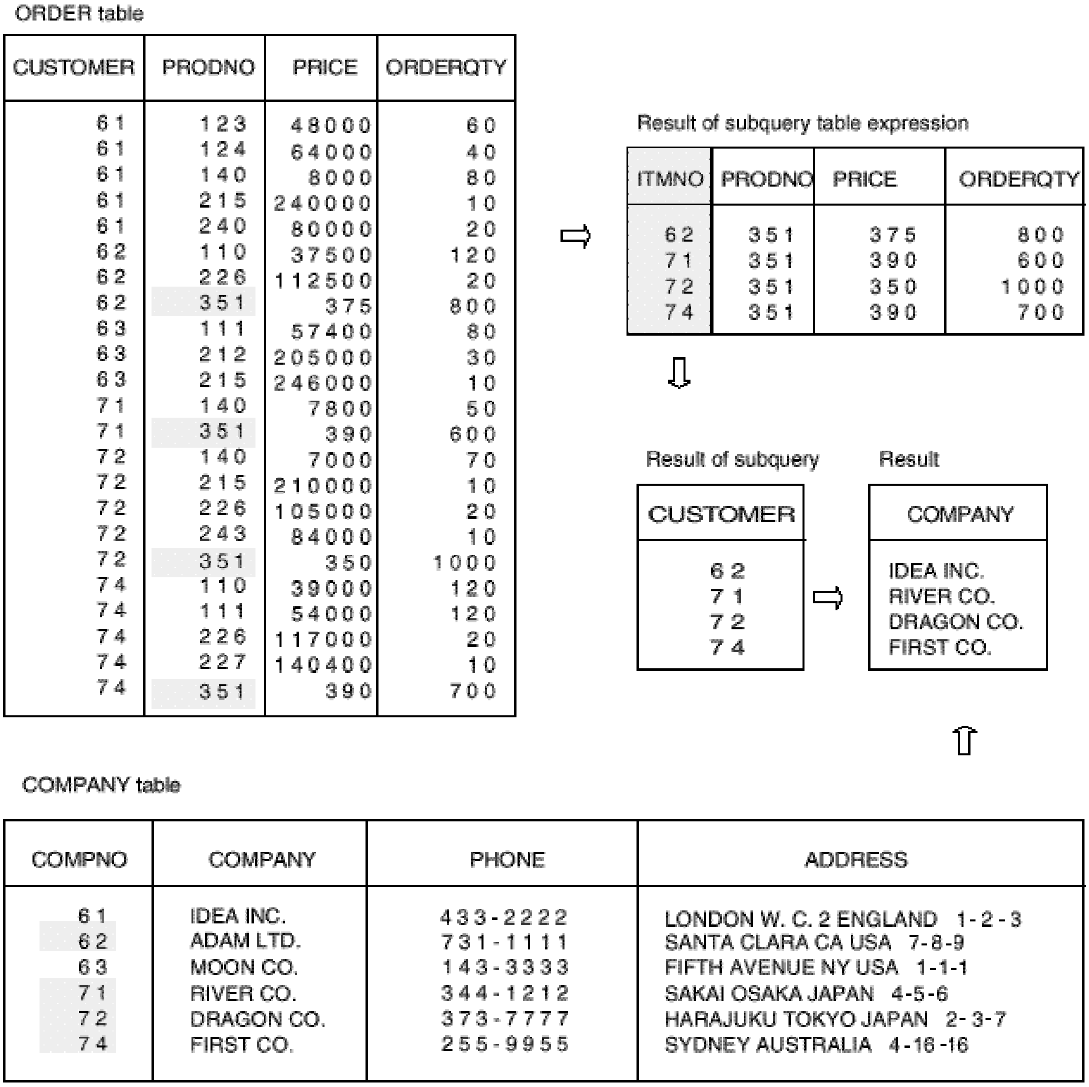
Example 2:

In this example, the names of companies that have ordered the product that has ITMNO (PRODNO) "351" are fetched.



```
SELECT COMPANY FROM STOCKS. COMPANY WHERE
COMPNO IN (SELECT CUSTOMER FROM STOCKS. ORDER WHERE PRODNO = 351)
```

[Figure: Example of using a subquery in the IN predicate]



ORDER table

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	123	48000	60
61	124	64000	40
61	140	8000	80
61	215	240000	10
61	240	80000	20
62	110	37500	120
62	226	112500	20
62	351	375	800
63	111	57400	80
63	212	205000	30
63	215	246000	10
71	140	7800	50
71	351	390	600
72	140	7000	70
72	215	210000	10
72	226	105000	20
72	243	84000	10
72	351	350	1000
74	110	39000	120
74	111	54000	120
74	226	117000	20
74	227	140400	10
74	351	390	700

Result of subquery table expression

ITMNO	PRODNO	PRICE	ORDERQTY
62	351	375	800
71	351	390	600
72	351	350	1000
74	351	390	700

Result of subquery

CUSTOMER
62
71
72
74

Result

COMPANY
IDEA INC.
RIVER CO.
DRAGON CO.
FIRST CO.

COMPANY table

COMPNO	COMPANY	PHONE	ADDRESS
61	IDEA INC.	433-2222	LONDON W. C. 2 ENGLAND 1-2-3
62	ADAM LTD.	731-1111	SANTA CLARA CA USA 7-8-9
63	MOON CO.	143-3333	FIFTH AVENUE NY USA 1-1-1
71	RIVER CO.	344-1212	SAKAI OSAKA JAPAN 4-5-6
72	DRAGON CO.	373-7777	HARAJUKU TOKYO JAPAN 2-3-7
74	FIRST CO.	255-9955	SYDNEY AUSTRALIA 4-16-16

■Comparing a set of values

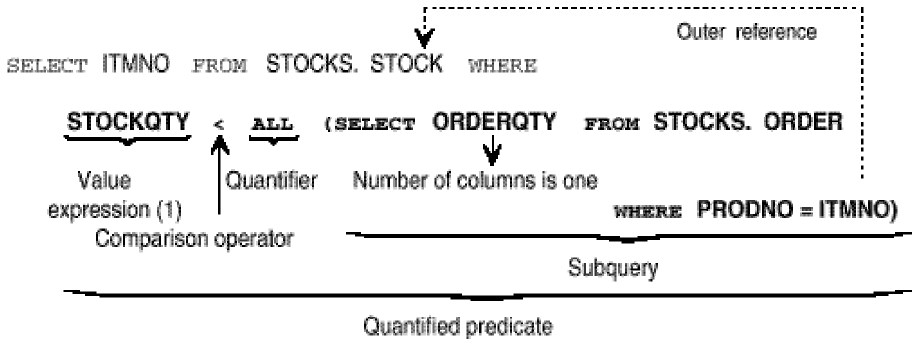
Use a quantified predicate as a search condition to specify comparison with a set of quantified values. Use a comparison operator and quantifier (ALL, SOME, or ANY) in specifying the quantified predicate.

The format of the subquery specification in parentheses is the same as that for a query specification. However, a table derived from a subquery can only have one column. This means that only one value expression (column name, function, and so on) can be specified on the left-hand side of the search condition.

A column that cannot be included in a table specified by a subquery (but can be included in a table specified by an outer subquery) can be specified in the search condition of the subquery. This subquery is called a subquery that has an outer reference. The following is an example of a subquery that has an outer reference specified in the quantified predicate:

Example 3:

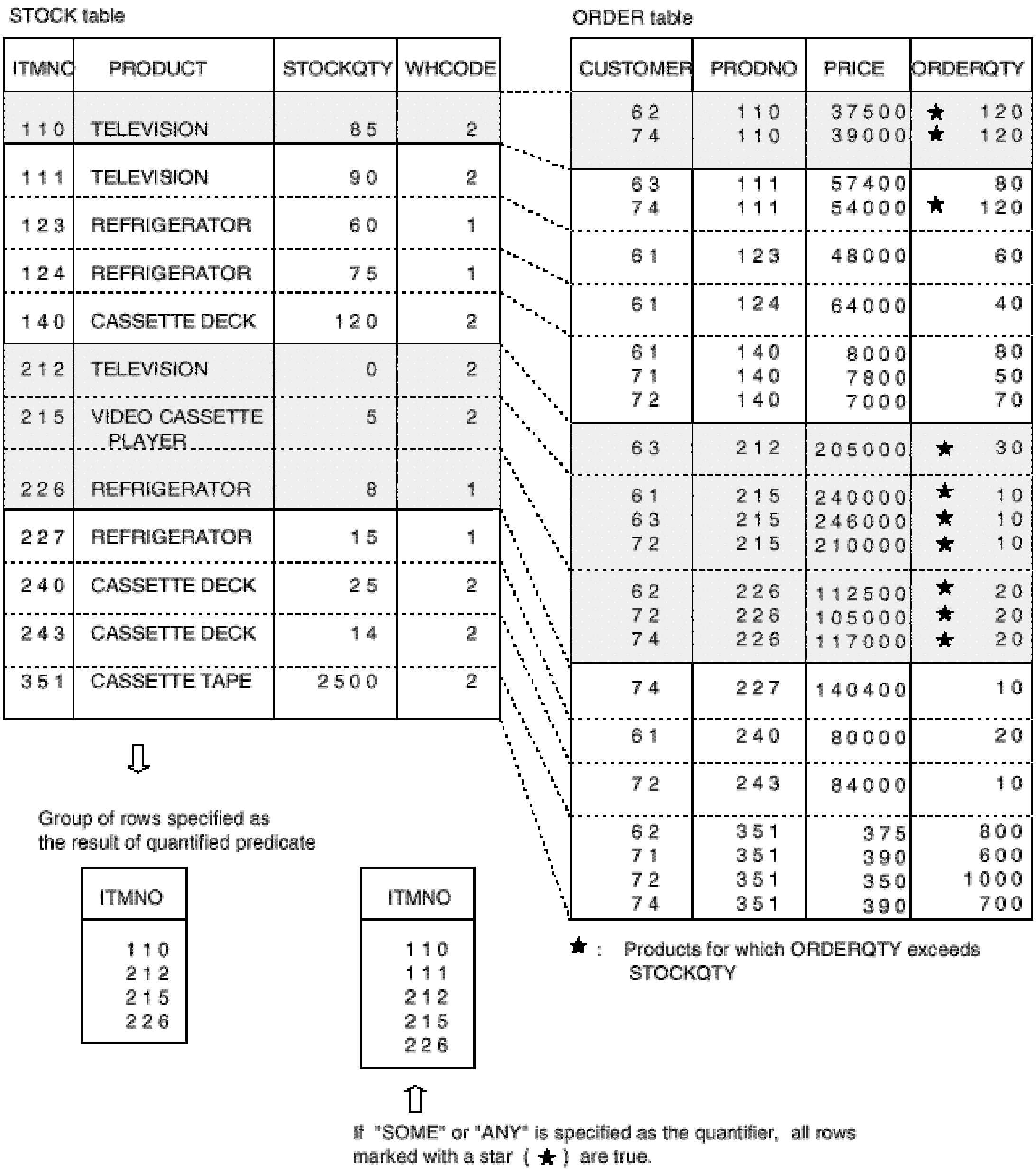
> In this example, values of ITMNO for which ORDERQTY for all CUSTOMER exceeds STOCKQTY are fetched. The column that cannot be included in the subquery table (ORDER table), that is, the column (ITMNO) that can be included in the outer query table (ORDER table), is specified in the search condition. Figure: Example of specifying a quantified predicate shows the retrieval results.



```
SELECT ITMNO FROM STOCKS. STOCK WHERE
STOCKQTY < ALL (SELECT ORDERQTY FROM STOCKS. ORDER
                            WHERE PRODNO = ITMNO)
```

Outer reference
Value expression (1)
Quantifier
Comparison operator
Number of columns is one
Subquery
Quantified predicate

[Figure: Example of specifying a quantified predicate]



STOCK table
ORDER table
Group of rows specified as the result of quantified predicate

STOCK table

ITMNO	PRODUCT	STOCKQTY	WHCODE
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
140	CASSETTE DECK	120	2
212	TELEVISION	0	2
215	VIDEO CASSETTE PLAYER	5	2
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETTE TAPE	2500	2

ORDER table

CUSTOMER	PRODNO	PRICE	ORDERQTY
62	110	37500	★ 120
74	110	39000	★ 120
63	111	57400	80
74	111	54000	★ 120
61	123	48000	60
61	124	64000	40
61	140	8000	80
71	140	7800	50
72	140	7000	70
63	212	205000	★ 30
61	215	240000	★ 10
63	215	246000	★ 10
72	215	210000	★ 10
62	226	112500	★ 20
72	226	105000	★ 20
74	226	117000	★ 20
74	227	140400	10
61	240	80000	20
72	243	84000	10
62	351	375	800
71	351	390	600
72	351	350	1000
74	351	390	700

★ : Products for which ORDERQTY exceeds STOCKQTY

Group of rows specified as the result of quantified predicate

ITMNO
110
212
215
226

ITMNO
110
111
212
215
226

If "SOME" or "ANY" is specified as the quantifier, all rows marked with a star (★) are true.

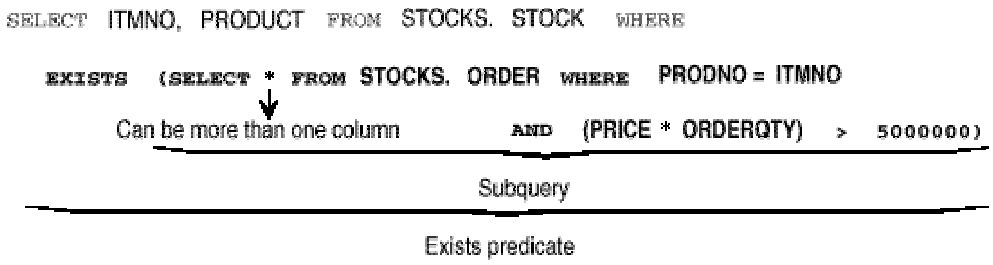
■Checking whether rows that satisfy certain conditions exist

The EXISTS predicate can be specified in a subquery. Whether rows are specified depends on whether or not the result of the subquery is null. In other words, the EXISTS predicate is used when the row count for the table derived from the result of the subquery can be 0. If the result of the subquery specified after EXISTS is not null, the EXISTS predicate is true. If the result of the subquery is null, the EXISTS predicate is false.

The format of the subquery specification in parentheses is the same as that for a query specification. The result of the subquery can be a table with two or more columns. The following is an example of specifying the EXISTS predicate:

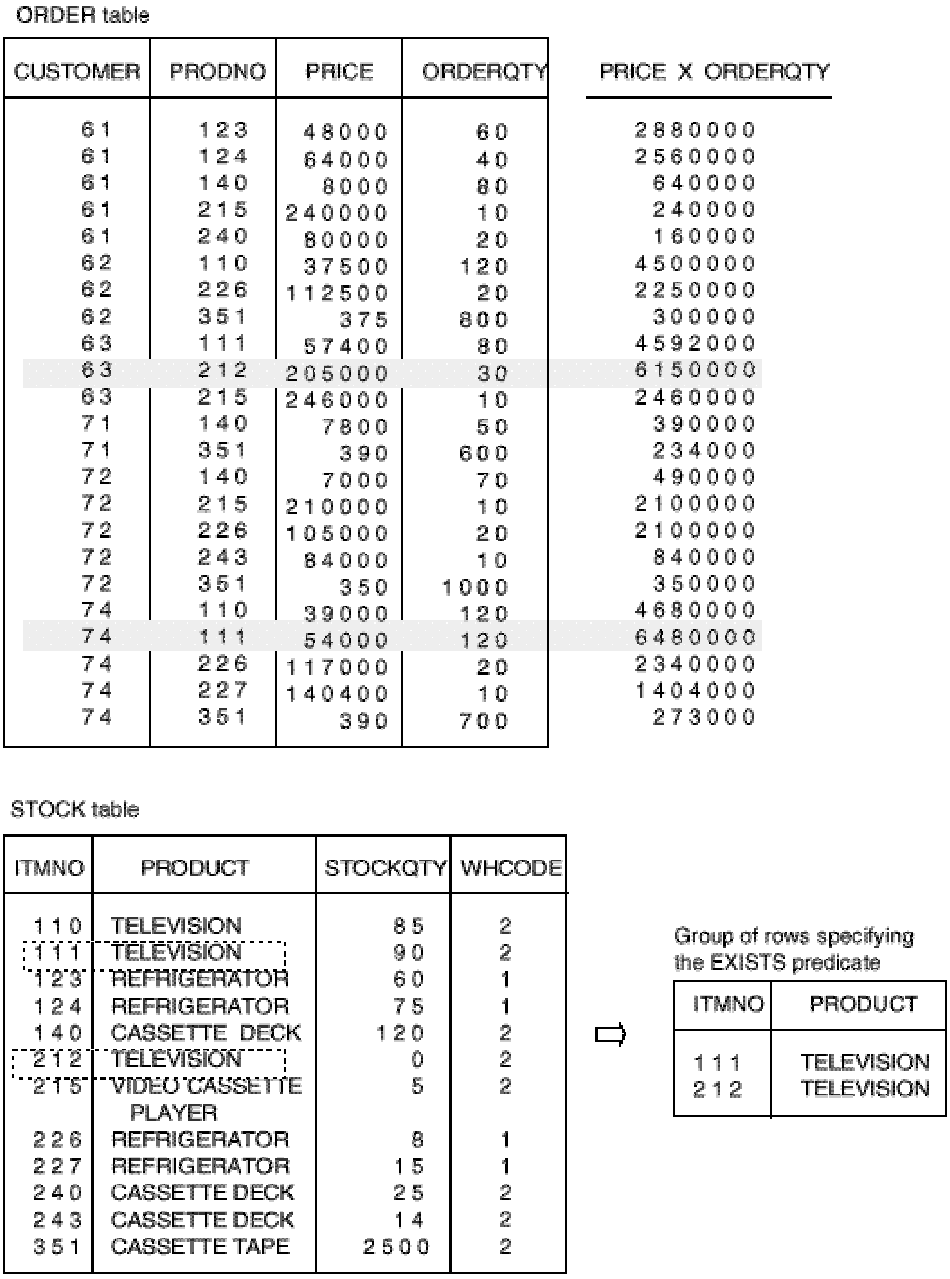
Example 4:

In this example, data for ITMNO and PRODUCT for products for which the value of orders is greater than 5,000,000 is fetched. The value of orders is obtained by PRICE x ORDERQTY.



```
SELECT ITMNO, PRODUCT FROM STOCKS. STOCK WHERE
EXISTS (SELECT * FROM STOCKS. ORDER WHERE PRODNO = ITMNO
                                   AND (PRICE * ORDERQTY) > 5000000)
```

[Figure: Example of specifying the EXISTS predicate]



ORDER table

CUSTOMER	PRODNO	PRICE	ORDERQTY	PRICE X ORDERQTY
61	123	48000	60	2880000
61	124	64000	40	2560000
61	140	8000	80	640000
61	215	240000	10	240000
61	240	80000	20	160000
62	110	37500	120	4500000
62	226	112500	20	2250000
62	351	375	800	300000
63	111	57400	80	4592000
63	212	205000	30	6150000
63	215	246000	10	2460000
71	140	7800	50	390000
71	351	390	600	234000
72	140	7000	70	490000
72	215	210000	10	2100000
72	226	105000	20	2100000
72	243	84000	10	840000
72	351	350	1000	350000
74	110	39000	120	4680000
74	111	54000	120	6480000
74	226	117000	20	2340000
74	227	140400	10	1404000
74	351	390	700	273000

STOCK table

ITMNO	PRODUCT	STOCKQTY	WHCODE
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
140	CASSETTE DECK	120	2
212	TELEVISION	0	2
215	VIDEO CASSETTE PLAYER	5	2
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETTE TAPE	2500	2

Group of rows specifying the EXISTS predicate

ITMNO	PRODUCT
111	TELEVISION
212	TELEVISION

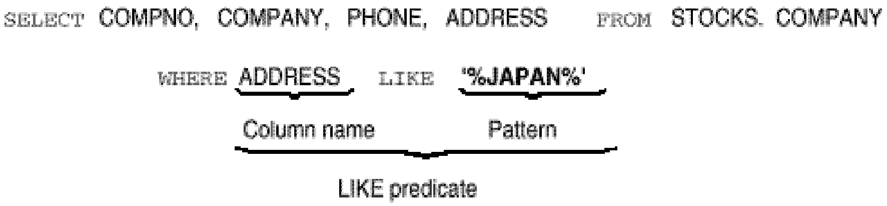
6.2.4 Checking whether values match a character-string pattern

Use the LIKE predicate to check whether data in a character-string type or national-language character-string type matches a certain character-string pattern. The result of the LIKE predicate is true if the specified pattern is contained in a column. The result is false if the pattern is not contained in any column. The result is undefined if the specified columns or pattern have a null value. Data in character-string or national-language character-string type, and blank characters in a pattern are treated as a single alphabetic or national-language character.

If the string or pattern specified in the LIKE predicate is a variable-length character string, data within the valid range is evaluated. The following is an example of specifying the LIKE predicate:

Example 1:

In this example, rows where "JAPAN" appears in the ADDRESS column of the COMPANY table are specified. Figure: Using the LIKE predicate to specify rows shows the result of this specification.



```
SELECT COMPNO, COMPANY, PHONE, ADDRESS FROM STOCKS. COMPANY
WHERE ADDRESS LIKE '%JAPAN%'
```

[Figure: Using the LIKE predicate to specify rows]

COMPANY table

COMPNO	COMPANY	PHONE	ADDRESS
61	IDEA INC.	433-2222	LONDON W.C. 2 ENGLAND 1-2-3
62	ADAM LTD.	731-1111	SANTA CLARA CA USA 7-8-9
63	MOON CO.	143-3333	FIFTH AVENUE NY USA 1-1-1
71	RIVER CO.	344-1212	SAKAI OSAKA JAPAN 4-5-6
72	DRAGON CO.	373-7777	HARAJUKU TOKYO JAPAN 2-3-7
74	FIRST CO.	255-9955	SYDNEY AUSTRALIA 4-16-16

⇓

Group of rows specified as a result of the LIKE predicate

COMPNO	COMPANY	PHONE	ADDRESS
71	RIVER CO.	344-1212	SAKAI OSAKA JAPAN 4-5-6
72	DRAGON CO.	373-7777	HARAJUKU TOKYO JAPAN 2-3-7

Use a character string, arbitrary string specifier, or arbitrary character specifier to represent the pattern. Table: Arbitrary string specifier and arbitrary character specifier explains the arbitrary string specifier and the arbitrary character specifier. When the pattern is a character string, use alphabetic characters to specify the pattern. When the pattern is in a national language, use national-language characters to specify the pattern.

[Table: Arbitrary string specifier and arbitrary character specifier]

Specifier	Symbol	Explanation
Arbitrary string specifier	% (Alphanumeric character) % (National-language character)	Represents any character string of zero or more characters
Arbitrary character specifier	- (Alphabetic character) _ (National-language character)	Represents any single character

■Examples of using arbitrary string specifier %

Examples of using arbitrary string specifier% are shown below. "Figure: Example of LIKE predicate specification (using an arbitrary string specifier)" shows which row is specified. Schema SCM contains the table TBL5 used in these examples. SCM.

Example 1:

Rows on which the value in column COL1 in table TBL5 is "ABC" are specified.

SELECT ... FROM SCM.TBL5 WHERE **COL1 LIKE 'ABC'**

Example 2:

In this example, rows for which the value in column COL1 in table TBL5 begins with "ABC" are specified.

SELECT ... FROM SCM.TBL5 WHERE **COL1 LIKE 'ABC%'**

Example 3:

Rows on which the value in column COL1 in table TBL5 begins with "ABC" are specified.

SELECT ... FROM SCM.TBL5 WHERE **COL1 LIKE '%ABC'**

Example 4:

Rows containing "ABC" at any position in column COL1 of table TBL5 are specified.

SELECT ... FROM SCM.TBL5 WHERE **COL1 LIKE '%ABC%'**

Example :5

Rows on which the value in column COL1 in table TBL5 begins with "AB" and ends with "C" are specified.

SELECT ... FROM SCM.TBL5 WHERE **COL1 LIKE 'AB%C'**

Example :6

Rows on which the value in column COL1 in table TBL5 begins with "ABC" and ends with a blank are specified.

SELECT ... FROM SCM.TBL5 WHERE **COL1 LIKE 'ABC% '**

Example 7:

Rows on which the value in column COL1 in table TBL5 begins with "ABC" are specified. To define the pattern to host variable HOST1 (fixed size of 6 bytes), specify "ABC%". Use the TRIM function to remove blanks at the end.

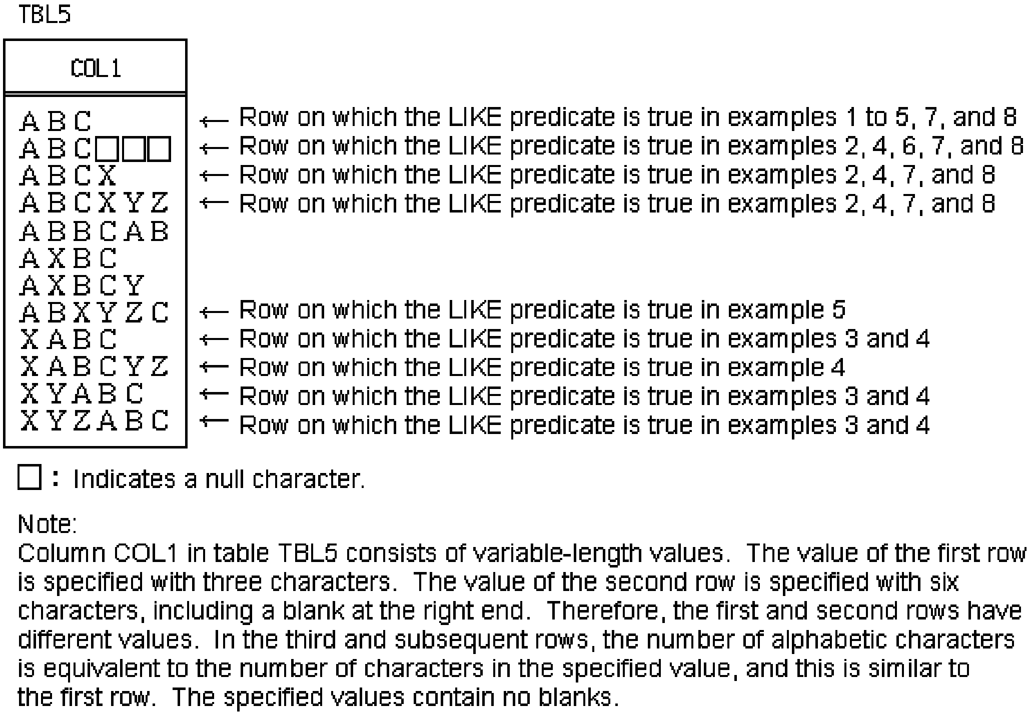
SELECT ... FROM SCM.TBL5 WHERE **COL1 LIKE TRIM(TRAILING :HOST1)**

Example 8:

Rows on which the value in column COL1 in table TBL5 begins with "ABC". To define the pattern to variable-length host variable HOST1, specify "ABC%".

SELECT ... FROM SCM.TBL5 WHERE **COL1 LIKE :HOST1**

[Figure: Example of LIKE predicate specification (using an arbitrary string specifier)]


TBL5

COL1	
ABC	← Row on which the LIKE predicate is true in examples 1 to 5, 7, and 8
ABC□□□	← Row on which the LIKE predicate is true in examples 2, 4, 6, 7, and 8
ABCX	← Row on which the LIKE predicate is true in examples 2, 4, 7, and 8
ABCXYZ	← Row on which the LIKE predicate is true in examples 2, 4, 7, and 8
ABBCAB	
AXBC	
AXBCY	
ABXYZC	← Row on which the LIKE predicate is true in example 5
XABC	← Row on which the LIKE predicate is true in examples 3 and 4
XABCYZ	← Row on which the LIKE predicate is true in example 4
XYABC	← Row on which the LIKE predicate is true in examples 3 and 4
XYZABC	← Row on which the LIKE predicate is true in examples 3 and 4

□ : Indicates a null character.

Note:
Column COL1 in table TBL5 consists of variable-length values. The value of the first row is specified with three characters. The value of the second row is specified with six characters, including a blank at the right end. Therefore, the first and second rows have different values. In the third and subsequent rows, the number of alphabetic characters is equivalent to the number of characters in the specified value, and this is similar to the first row. The specified values contain no blanks.

■Examples of using arbitrary string specifier "_"

Examples of using arbitrary string specifier "_" are shown below. The figure with the example of "Figure: LIKE predicate specification (using an arbitrary string specifier)" shows which row is specified.

Example 1:

Rows of four characters for which the value in column COL1 in table TBL5 begins with "ABC" and ends with any character are specified.

SELECT ... FROM SCM.TBL5 WHERE **COL1 LIKE 'ABC_'**

Example 2:

Row of six characters for which the value in column COL1 in table TBL5 begins with "ABC" and ends with any three characters are specified.

SELECT ... FROM SCM.TBL5 WHERE **COL1 LIKE 'ABC___'**

Example 3:

Rows of four characters for which the value in column COL1 in table TBL5 begins with any character and ends with "ABC" are specified.

SELECT ... FROM SCM.TBL5 WHERE **COL1 LIKE '_ABC'**

Example 4:

Rows of five characters for which the value in column COL1 in TBL5 begins with any two characters and ends with "ABC" are specified.

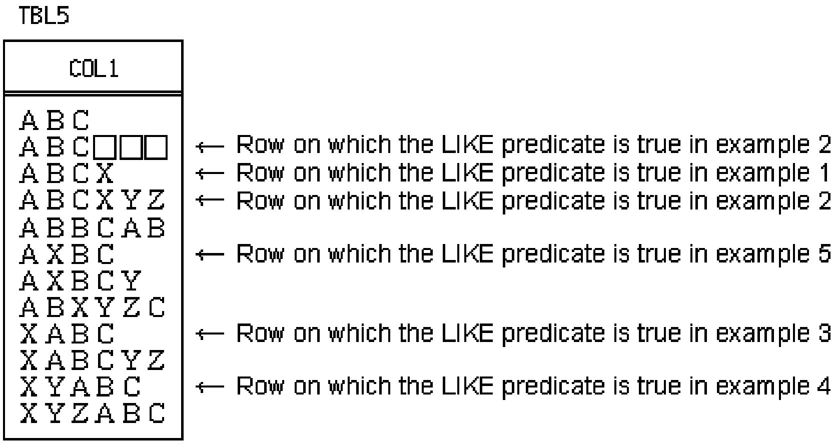
SELECT ... FROM SCM.TBL5 WHERE **COL1 LIKE '__ABC'**

Example 5:

Rows of four characters for which the value in column COL1 in TBL5 begins with any character followed by "A" and ends with "BC".

SELECT ... FROM SCM.TBL5 WHERE **COL1 LIKE 'A_BC'**

[Figure: Example of LIKE predicate specification (using an arbitrary string specifier)]



TBL5

COL1	
ABC	
ABC□□□	← Row on which the LIKE predicate is true in example 2
ABCX	← Row on which the LIKE predicate is true in example 1
ABCXYZ	← Row on which the LIKE predicate is true in example 2
ABBCAB	
AXBC	← Row on which the LIKE predicate is true in example 5
AXBCY	
ABXYZC	
XABC	← Row on which the LIKE predicate is true in example 3
XABCYZ	
XYABC	← Row on which the LIKE predicate is true in example 4
XYZABC	

□ : Indicates a null character.

Note:
Column COL1 in table TBL5 consists of variable-length values. The value of the first row is specified with three characters. The value of the second row is specified with six characters, including a blank at the right end. Therefore, the first and second rows have different values. In the third and subsequent rows, the number of alphabetic characters is equivalent to the number of characters in the specified value, and this is similar to the first row. The specified values contain no blanks.

■Escape characters

In a pattern specification in the LIKE predicate, the percent and underscore signs specify the location of the arbitrary characters. If the percent and underscore signs themselves are to be searched, specify an escape character. Examples of specifying the escape character follow. Figure: Example of specifying a LIKE predicate specification using escape characters shows the characters specified in the examples.

Example 6:

In this example, rows for which the value in column COL2 in table TBL5 is "ABC%" are specified.

```
SELECT ... FROM SCM. TBL5 WHERE COL2 LIKE  'ABC#%' ESCAPE '#'
```

Escape character

The character following the word "ESCAPE" is the escape character. In a character pattern, the escape character is specified immediately before the percent or underscore sign to be searched. Any character can be specified as the escape character.

Example 7:

In this example, rows for which the value in column COL2 in table TBL5 table is "ABC%%" are specified.

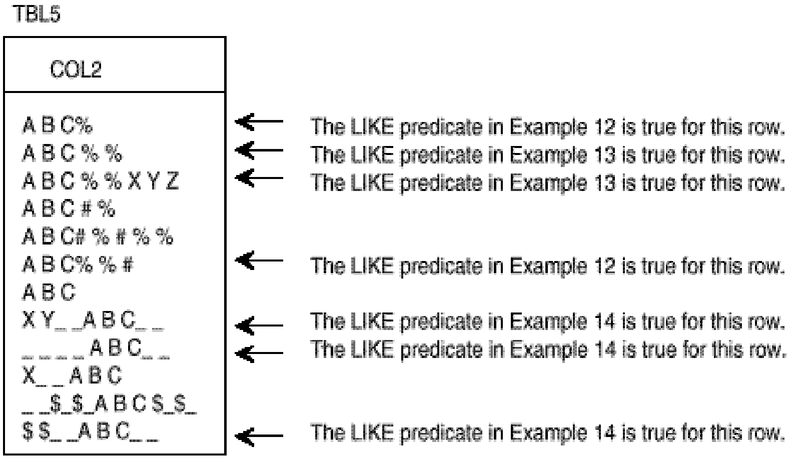
```
SELECT ... FROM SCM. TBL5 WHERE COL2 LIKE ‘ABC#%#%’ ESCAPE ‘#’
```

Example 8:

In this example, rows for which the value in column COL2 in table TBL5 begins with any two characters and ends with "_ _ABC_ _" are specified.

```
SELECT ... FROM SCM. TBL5 WHERE COL2 LIKE '__$_$_ABC$_$_' ESCAPE '$'
```

[Figure: Example of specifying a LIKE predicate specification using escape characters]



TBL5

COL2	
ABC%	The LIKE predicate in Example 12 is true for this row.
ABC%%	The LIKE predicate in Example 13 is true for this row.
ABC%%XYZ	The LIKE predicate in Example 13 is true for this row.
ABC#%	
ABC#%#%%	
ABC%%#	The LIKE predicate in Example 12 is true for this row.
ABC	
XY__ABC__	The LIKE predicate in Example 14 is true for this row.
____ABC__	The LIKE predicate in Example 14 is true for this row.
X__ABC	
__$_$_ABC$_$_	
$$__ABC__	The LIKE predicate in Example 14 is true for this row.

Note: The values in column COL2 in table TBL5 are variable-length values. The value in row 1 has four characters. The values from row 2 are calculated in the same way as in row 1. That is, the number of alphabetic characters, percent, pound, and dollar signs, and underscores is the number of characters.

6.3 Manipulating Numeric Data

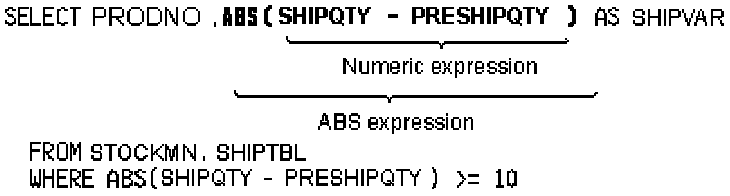
Users can manipulate numeric data to obtain absolute values, minimum integer values, maximum integer values, and other values. The table below lists the processing used to manipulate numeric data.

[Table: Processing for manipulating numeric data]

Processing name	Function
ABS expression	Obtains the absolute value of a specified numeric expression.
CELL expression	Obtains the minimum integer value or higher for a specified numeric expression.
FLOOR expression	Obtains the maximum integer value or lower for a specified numeric expression.
ROUND expression	Rounds off a numeric expression to a specified position. The specified position means the significant decimal digit where the rounding off takes effect.
TRUNC expression	Truncates a numeric expression to a truncation position. The truncation position means the significant decimal digit where the truncation takes effect.

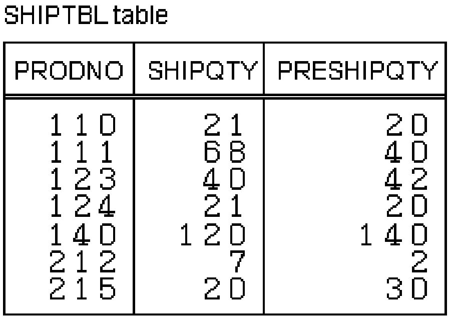
Example 1:

For products with a high variation of shipment quantity (assuming that the difference between the preceding and current shipment quantities is 10 or greater), the corresponding product numbers are obtained from the SHIPTBL table.



```
SELECT PRODNO ,ABS(SHIPQTY - PRESHIPQTY ) AS SHIPVAR
  FROM STOCKMN. SHIPTBL
  WHERE ABS(SHIPQTY - PRESHIPQTY ) >= 10
```

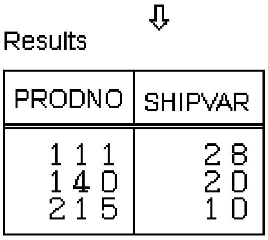
Numeric expression
ABS expression



SHIPTBL table

PRODNO	SHIPQTY	PRESHIPQTY
110	21	20
111	68	40
123	40	42
124	21	20
140	120	140
212	7	2
215	20	30

⇩

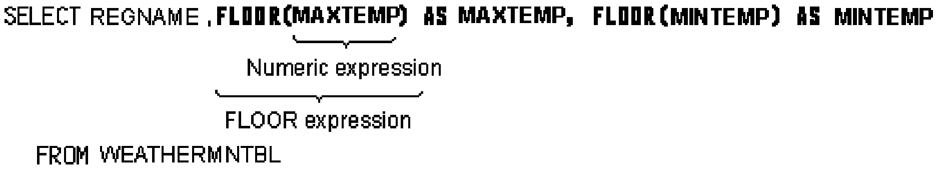


Results

PRODNO	SHIPVAR
111	28
140	20
215	10

Example 2:

The maximum and minimum temperatures in each region are rounded down to the next integers.



```
SELECT REGNAME ,FLOOR(MAXTEMP) AS MAXTEMP, FLOOR(MINTEMP) AS MINTEMP
  FROM WEATHERMNTBL
```

Numeric expression
FLOOR expression

WEATHERMNTBL table

REGCODE	REGNAME	MAXTEMP	MAXTEMPDATE	MINTEMP	MINTEMPDATE
1111	SAPPORO □□□	27.255	1978-07-14	-21.235	1994-01-11
2222	SENDAI □□□	30.000	1999-08-01	-8.500	1983-02-02
3333	TOKYO □□□	35.554	1984-07-23	-0.456	1988-12-23
4444	YOKOHAMA □□□	34.123	1988-08-01	0.789	1967-01-04
5555	OSAKA □□□	32.987	1995-08-11	6.357	1979-02-15
6666	HIROSHIMA □□□	31.567	1997-09-01	11.258	1990-02-06
7777	HAKATA □□□	37.525	1985-06-30	17.369	1995-12-27

□ : Indicates an em-size null character.

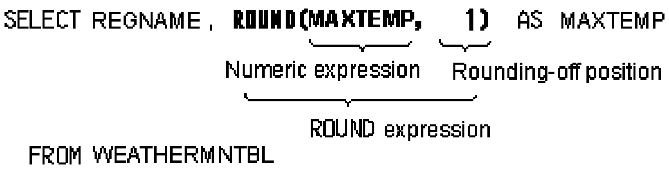
⇩

Results

REGNAME	MAXTEMP	MINTEMP
SAPPORO □□□	27.000	-22.000
SENDAI □□□	30.000	-9.000
TOKYO □□□	35.000	-1.000
YOKOHAMA □□□	34.000	0.000
OSAKA □□□	32.000	6.000
HIROSHIMA □□□	31.000	11.000
HAKATA □□□	37.000	17.000

Example 3:

The maximum temperature in each region is rounded off to two decimal places.



```
SELECT REGNAME, ROUND(MAXTEMP,  1) AS MAXTEMP
  FROM WEATHERMNTBL
```

WEATHERMNTBL table

REGCODE	REGNAME	MAXTEMP	MAXTEMPDATE	MINTEMP	MINTEMPDATE
1111	SAPPORO □□□	27.255	1978-07-14	-21.235	1994-01-11
2222	SENDAI □□□	30.000	1999-08-01	-8.500	1983-02-02
3333	TOKYO □□□	35.554	1984-07-23	-0.456	1988-12-23
4444	YOKOHAMA □□□	34.123	1988-08-01	0.789	1967-01-04
5555	OSAKA □□□	32.987	1995-08-11	6.357	1979-02-15
6666	HIROSHIMA □□□	31.567	1997-09-01	11.258	1990-02-06
7777	HAKATA □□□	37.525	1985-06-30	17.369	1995-12-27

□ : Indicates an em-size null character.

⇩

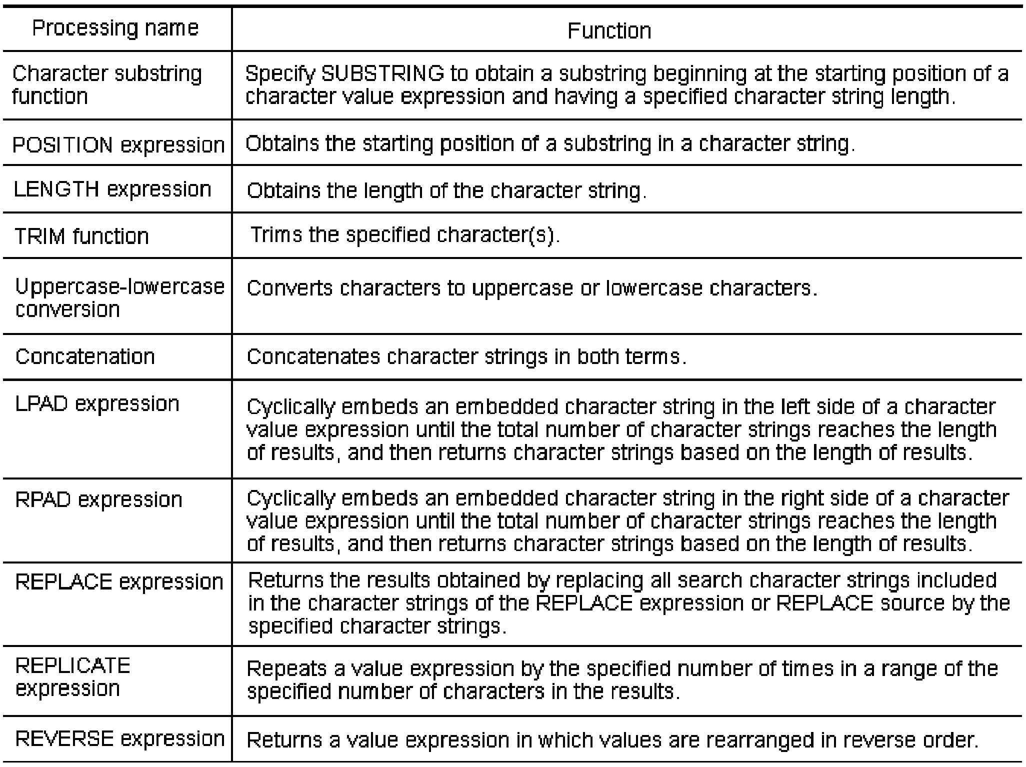
Results

REGNAME	MAXTEMP
SAPPORO □□□	27.300
SENDAI □□□	30.000
TOKYO □□□	35.600
YOKOHAMA □□□	34.100
OSAKA □□□	33.000
HIROSHIMA □□□	31.600
HAKATA □□□	37.500

6.4 Manipulating Character String Data

When manipuling table data, users can manipulate and join character strings. The table below lists the processing for manipulating parts of character strings.

[Table: Processing for manipulating parts of character strings]



Processing name	Function
Character substring function	Specify SUBSTRING to obtain a substring beginning at the starting position of a character value expression and having a specified character string length.
POSITION expression	Obtains the starting position of a substring in a character string.
LENGTH expression	Obtains the length of the character string.
TRIM function	Trims the specified character(s).
Uppercase-lowercase conversion	Converts characters to uppercase or lowercase characters.
Concatenation	Concatenates character strings in both terms.
LPAD expression	Cyclically embeds an embedded character string in the left side of a character value expression until the total number of character strings reaches the length of results, and then returns character strings based on the length of results.
RPAD expression	Cyclically embeds an embedded character string in the right side of a character value expression until the total number of character strings reaches the length of results, and then returns character strings based on the length of results.
REPLACE expression	Returns the results obtained by replacing all search character strings included in the character strings of the REPLACE expression or REPLACE source by the specified character strings.
REPLICATE expression	Repeats a value expression by the specified number of times in a range of the specified number of characters in the results.
REVERSE expression	Returns a value expression in which values are rearranged in reverse order.

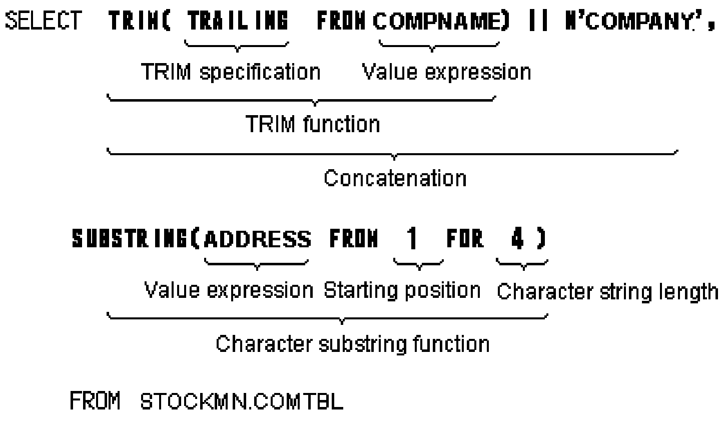
Example 1:

Company names and addresses are fetched from the company table. For each company name, a null character at the end is removed, "Co., Ltd.," is concatenated, and then only the metropolis and district names are fetched from the address.

Users can fetch a part of a specific character string and join a value to a character string in a specific column. The TRIM function removes characters from the beginning of a value expression if LEADING is specified in the TRIM function; it removes them from the end of a value expression if TRAILING is specified; and it removes them from both ends of a value expression if BOTH is specified.

To concatenate character strings, join them to concatenate operator "||". To do this, the data type of the column must be the character type. The data types of both terms to be concatenated must be the character type, and they must be comparable. For details on the comparable data types, see Table: Comparable Data Types.

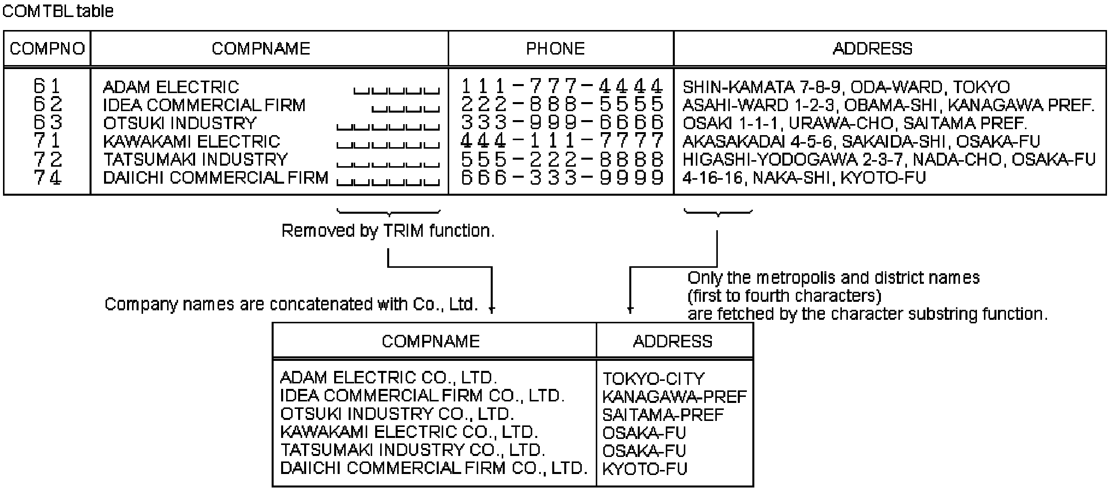
In the character substring function, SUBSTRING is specified to obtain a substring of a specific character string length beginning from the starting position in a value expression.



```
SELECT  TRIM( TRAILING  FROM COMPNAME) || N'COMPANY',
        SUBSTRING(ADDRESS FROM 1 FOR 4 )
        FROM  STOCKMN.COMTBL
```

TRIM specification
Value expression
TRIM function
Concatenation
Value expression
Starting position
Character string length
Character substring function

[Figure: Example of combining character value functions to fetch data]



COMTBL table

COMPNO	COMPNAME	PHONE	ADDRESS
61	ADAM ELECTRIC ␣␣␣␣␣	111-777-4444	SHIN-KAMATA 7-8-9, ODA-WARD, TOKYO
62	IDEA COMMERCIAL FIRM ␣␣␣␣	222-888-5555	ASAHI-WARD 1-2-3, OBAMA-SHI, KANAGAWA PREF.
63	OTSUKI INDUSTRY ␣␣␣␣␣␣	333-999-6666	OSAKI 1-1-1, URAWA-CHO, SAITAMA PREF.
71	KAWAKAMI ELECTRIC ␣␣␣␣␣␣	444-111-7777	AKASAKADAI 4-5-6, SAKAIDA-SHI, OSAKA-FU
72	TATSUMAKI INDUSTRY ␣␣␣␣␣␣	555-222-8888	HIGASHI-YODOGAWA 2-3-7, NADA-CHO, OSAKA-FU
74	DAIICHI COMMERCIAL FIRM ␣␣␣␣␣␣	666-333-9999	4-16-16, NAKA-SHI, KYOTO-FU

Removed by TRIM function.

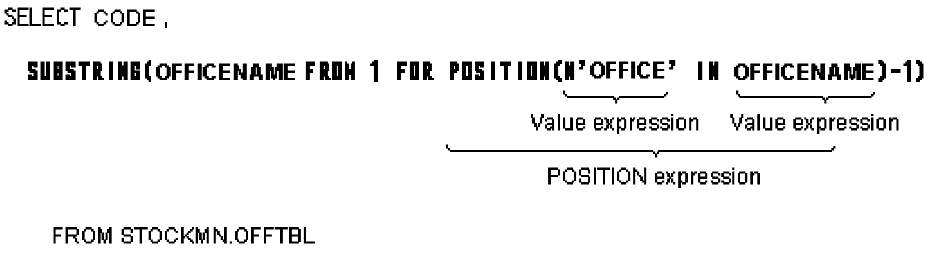
Company names are concatenated with Co., Ltd.

Only the metropolis and district names
(first to fourth characters)
are fetched by the character substring function.

COMPNAME	ADDRESS
ADAM ELECTRIC CO., LTD.	TOKYO-CITY
IDEA COMMERCIAL FIRM CO., LTD.	KANAGAWA-PREF
OTSUKI INDUSTRY CO., LTD.	SAITAMA-PREF
KAWAKAMI ELECTRIC CO., LTD.	OSAKA-FU
TATSUMAKI INDUSTRY CO., LTD.	OSAKA-FU
DAIICHI COMMERCIAL FIRM CO., LTD.	KYOTO-FU

Example 2:

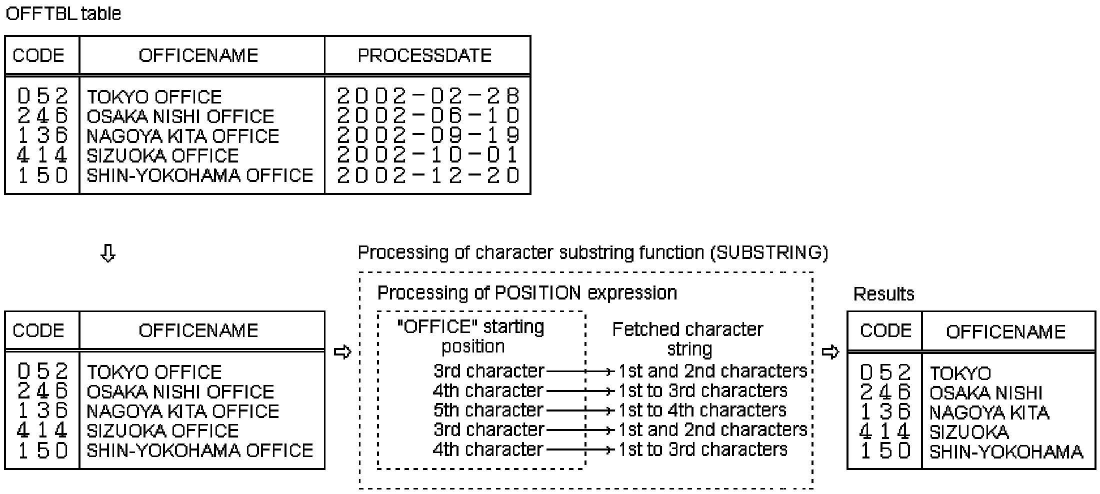
Codes and office names are fetched from the office list table, and character string "OFFICE" is removed from each office name.



```
SELECT CODE,
   SUBSTRING(OFFICENAME FROM 1 FOR POSITION(N'OFFICE' IN OFFICENAME)-1)
    FROM STOCKMN.OFFTBL
```

Value expression
Value expression
POSITION expression

[Figure: Example of combining a character substring function with a numeric function to fetch data]



OFFTBL table

CODE	OFFICENAME	PROCESSDATE
052	TOKYO OFFICE	2002-02-28
246	OSAKA NISHI OFFICE	2002-06-10
136	NAGOYA KITA OFFICE	2002-09-19
414	SIZUOKA OFFICE	2002-10-01
150	SHIN-YOKOHAMA OFFICE	2002-12-20

CODE	OFFICENAME
052	TOKYO OFFICE
246	OSAKA NISHI OFFICE
136	NAGOYA KITA OFFICE
414	SIZUOKA OFFICE
150	SHIN-YOKOHAMA OFFICE

Processing of character substring function (SUBSTRING)

Processing of POSITION expression

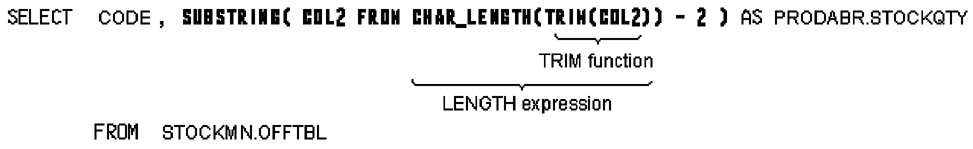
"OFFICE" starting position	Fetched character string
3rd character	1st and 2nd characters
4th character	1st to 3rd characters
5th character	1st to 4th characters
3rd character	1st and 2nd characters
4th character	1st to 3rd characters

Results

CODE	OFFICENAME
052	TOKYO
246	OSAKA NISHI
136	NAGOYA KITA
414	SIZUOKA
150	SHIN-YOKOHAMA

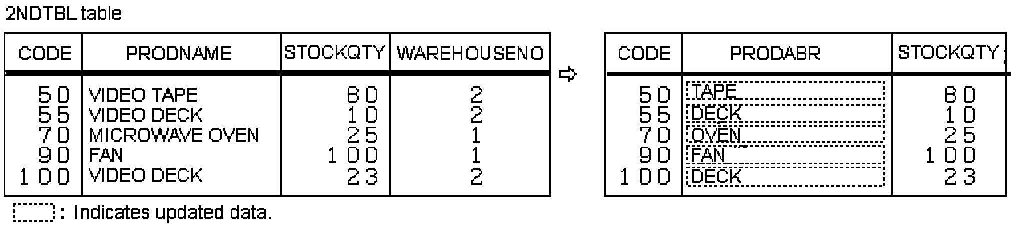
Example 3:

The last three characters are fetched from the product name character string in 2NDTBL.



```
SELECT   CODE , SUBSTRING( COL2 FROM CHAR_LENGTH(TRIM(COL2)) - 2 ) AS PRODABR.STOCKQTY
     FROM   STOCKMN.OFFTBL
```

[Figure: Example of combining a character substring function with a numeric function to update data]



2NDTBL table

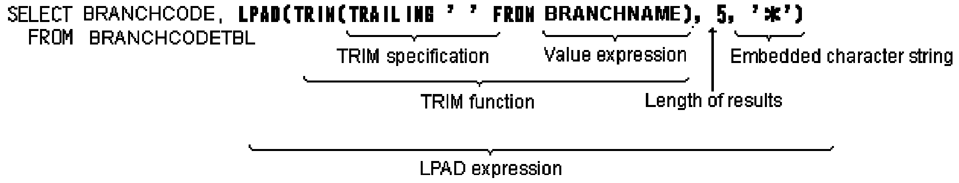
CODE	PRODNAME	STOCKQTY	WAREHOUSENO
50	VIDEO TAPE	80	2
55	VIDEO DECK	10	2
70	MICROWAVE OVEN	25	1
90	FAN	100	1
100	VIDEO DECK	23	2

CODE	PRODABR	STOCKQTY
50	TAPE	80
55	DECK	10
70	OVEN	25
90	FAN	100
100	DECK	23

: Indicates updated data.

Example 4:

An asterisk "*" is embedded on the left end of each branch name. The entire length is five characters.



```
SELECT BRANCHCODE, LPAD(TRIM(TRAILING ' ' FROM BRANCHNAME), 5, '*')
  FROM  BRANCHCODETBL
```

Branch code table

BRANCHCODE	BRANCHNAME	BRANCHADDRESS	PHONE
111	SAPPORO	NISHI 1-2, ODORI, CHUO-WARD, SAPPORO-CHI	011-999-9999
222	SENDAI	ICHIBAN-CHO 1-2-3, AOBA-WARD, SENDAI-SHI	022-111-1111
333	ROPPONGI	ROPPONGI 7-8-9, MINATO-WARD, TOKYO	03-3123-9876
444	SHIN-YOKOHAMA	SHIN-YOKOHAMA 1-23-45, KOHOKU-WARD, YOKOHAMA-SHI	045-987-6543
555	OSAKA	KITAHAMA 8-9, CHUO-WARD, OSAKA-SHI	06-6123-4567
666	HAKATA	HAKATAEKIMAE 1-2-3, HAKATA-WARD, FUKUOKA-SHI	092-987-1234

⇩

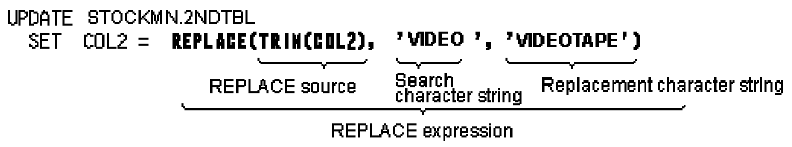
Results

111	＊＊＊SAPPORO
222	＊＊＊SENDAI
333	＊＊ ROPPONGI
444	＊＊ SHIN-YOKOHAMA
555	＊＊＊OSAKA
666	＊＊＊HAKATA

1 A column created from the results of LPAD(TRIM(TRAILING ' ' FROM BRANCHNAME), 5, '').

Example 5:

Character string "VIDEO" in COL2 of table 2NDTBL is replaced with "VIDEOTAPE".



```
UPDATE  STOCKMN.2NDTBL
  SET  COL2 =  REPLACE(TRIM(COL2),  'VIDEO ', 'VIDEOTAPE')
```

2NDTBL table

COL1	COL2	COL3	COL4
50	VIDEOTAPE	80	2
55	VIDEO DECK	10	2
70	MICROWAVE OVEN	25	1
90	FAN	100	1
100	VIDEO DECK	23	2

⇩

Results

COL1	COL2	COL3	COL4
50	VIDEOTAPE	80	2
55	VIDEO DECK	10	2
70	MICROWAVE OVEN	25	1
90	FAN	100	1
100	VIDEO DECK	23	2

6.5 Manipulating Date Data

Users can manipulate table data to calculate dates and times, convert date-time value expression data to characters, and convert character-type data to DATE-type data. The table below lists the processing for manipulating date data.

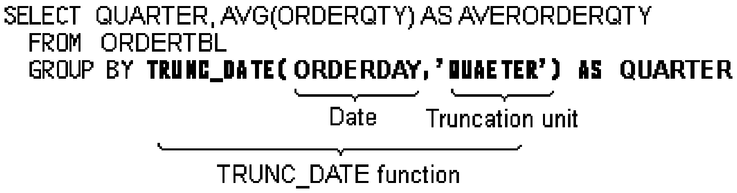
[Table: Processing for manipulating date data]

Processing name	Function
CURRENT DATE value function	Returns the current date.
CURRENT TIME value function	Returns the current time.
CURRENT TIMESTAMP value function	Returns the current timestamp.
ROUND_DATE function	Returns the date rounded off for each year, month, or quarter.
TRUNC_DATE function	Returns the date truncated for each year, month, or quarter.
SPAN_DATE function	Returns a time interval between dates.
ADD_DATE function	Returns the date-time value to the DATE type as the results of adding the time interval expressed by a numeric attribute to the date-type data.
LAST_DAY function	Returns the last date of a month.
CNV_CHAR function	Converts date-time value expression data to character-type data.
CNV_DATE function	Converts character-type data to DATE-type data.

Example 1:

The order quantity for each quarter is obtained from the order table.

By using the TRUNC_DATE function, the date can be truncated for each year (January 1 on the year), month (1st day of the month), or quarter.



```
SELECT  QUARTER, AVG(ORDERQTY) AS AVERORDERQTY
  FROM  ORDERTBL
  GROUP BY TRUNC_DATE(ORDERDAY,'QUAETER') AS QUARTER
```

ORDERTBL table

CLIENT	ORDERNO	ORDERQTY	ORDERDAY	DELIVERYDAY
61	123	60	2002-07-14	2002-10-20
62	124	40	2002-08-01	2002-11-20
63	140	80	2002-07-10	2002-07-20
71	215	10	2002-08-01	2002-11-20
72	240	20	2002-05-11	2002-08-20
74	110	120	2002-09-01	2002-12-20
74	226	20	2002-06-30	2002-09-20

⇩

Table derived from GROUP BY clause

CLIENT	ORDERNO	ORDERQTY	ORDERDAY	DELIVERYDAY
72	240	20	2002-05-11	2002-08-20
74	226	20	2002-06-30	2002-09-20
61	123	60	2002-07-14	2002-10-20
62	124	40	2002-08-01	2002-11-20
63	140	80	2002-07-10	2002-07-20
71	215	10	2002-08-01	2002-11-20
74	110	120	2002-09-01	2002-12-20

One group:
April 1 to June 30 of a year

One group:
July 1 to September 30 of a year

⇩

Results

QUARTER	AVERORDERQTY
2002-04-01	20
2002-07-01	62

Example 2:

The number of days that have been elapsed from the order date to the current date is obtained from the ORDERTBL table. The current date is 2000-08-10.

By using the SPAN_DATE function, the difference (time interval) between dates can be obtained, where YEAR, MONTH, or DAY is used as the unit for the time interval. Users can also obtain the number of times the year, month, or day changes between dates.

```
SELECT ORDERNO, SPAN_DATE(ORDERDAY,CURRENT_DATE,'DAY') AS ELAPSEDDAYS
                          Date      Date     Unit for time interval
                          SPAN_DATE function
  FROM ORDERTBL
  WHERE DELIVERYDAY IS NULL
```

ORDERTBL table

CLIENT	ORDERNO	ORDERQTY	ORDERDAY	DELIVERYDAY
61	123	60	2002-07-14	2002-10-20
62	124	40	2002-08-01	
63	140	80	2002-07-10	2002-07-20
71	215	10	2002-08-01	2002-11-20
72	240	20	2002-05-11	
74	110	120	2002-09-01	2002-12-20
74	226	20	2002-06-30	2002-09-20

NULL

NULL

⇩

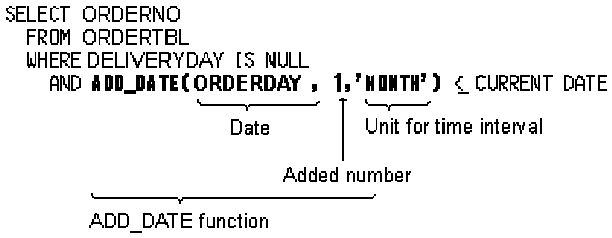
Results

ORDERNO	ELAPSEDDAYS
124	9
240	91

Example 3:

For orders that are one month old or older in the ORDERTBL table, the corresponding order numbers are obtained from the order table according to the specified order day. The current date is 2000-08-01.

By using the ADD_DATE function, dates can be added, where YEAR, MONTH, or DAY is used as the unit of addition for the time interval. The added number can be defined as a number of years, months, or days.



```
SELECT ORDERNO
  FROM ORDERTBL
  WHERE DELIVERYDAY IS NULL
    AND ADD_DATE(ORDERDAY , 1,'MONTH') <= CURRENT DATE
```

ORDERTBL table

CLIENT	ORDERNO	ORDERQTY	ORDERDAY	DELIVERYDAY
61	123	60	2002-07-14	2002-10-20
62	124	40	2002-08-01	
63	140	80	2002-07-10	2002-07-20
71	215	10	2002-08-01	2002-11-20
72	240	20	2002-05-11	
74	110	120	2002-09-01	2002-12-20
74	226	20	2002-06-30	2002-09-20

NULL

NULL

⇩

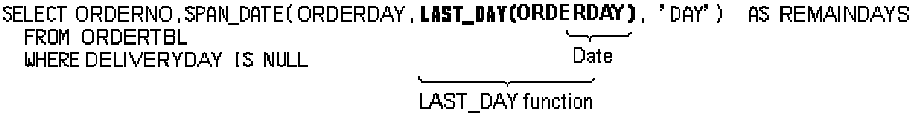
Results

ORDERNO
240

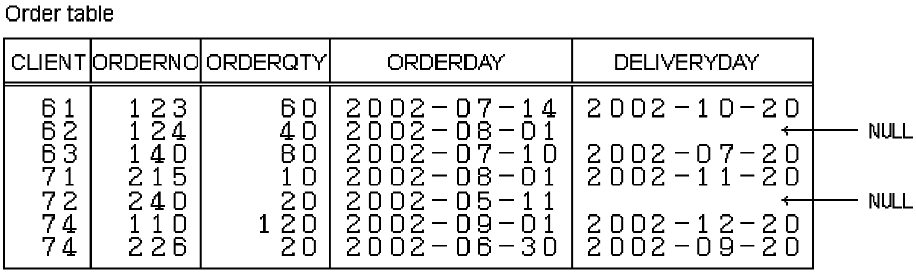
Example 4:

A number of days ranging from the order day until the end of month is obtained.

The last date of the month can be obtained using the LAST_DAY function.



```
SELECT ORDERNO,SPAN_DATE(ORDERDAY,LAST_DAY(ORDERDAY), 'DAY') AS REMAINDAYS
  FROM ORDERTBL
  WHERE DELIVERYDAY IS NULL
```



Order table

CLIENT	ORDERNO	ORDERQTY	ORDERDAY	DELIVERYDAY
61	123	60	2002-07-14	2002-10-20
62	124	40	2002-08-01	
63	140	80	2002-07-10	2002-07-20
71	215	10	2002-08-01	2002-11-20
72	240	20	2002-05-11	
74	110	120	2002-09-01	2002-12-20
74	226	20	2002-06-30	2002-09-20

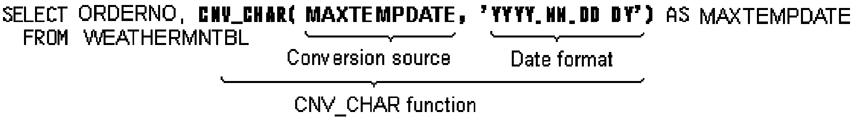
⇩

Results

ORDERNO	REMAINDAYS
124	30
240	20

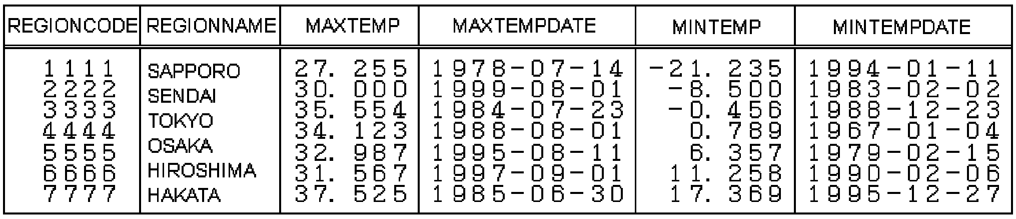
Example 5:

The maximum temperature date data in the WEATHERMNTBL table is converted to a character string in the "YYYY.MM.DD DY" format. The date type can be converted to the character type using the CNV_CHAR function. The conversion is performed in the date format.



```
SELECT ORDERNO, CNV_CHAR( MAXTEMPDATE, 'YYYY.MM.DD DY') AS MAXTEMPDATE
  FROM WEATHERMNTBL
```

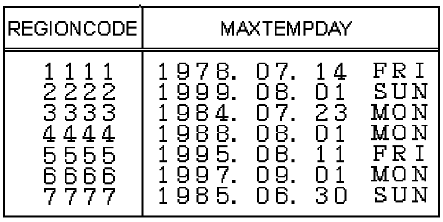
WEATHERMNTBL table



REGIONCODE	REGIONNAME	MAXTEMP	MAXTEMPDATE	MINTEMP	MINTEMPDATE
1111	SAPPORO	27.255	1978-07-14	-21.235	1994-01-11
2222	SENDAI	30.000	1999-08-01	-8.500	1983-02-02
3333	TOKYO	35.554	1984-07-23	-0.456	1988-12-23
4444	OSAKA	34.123	1988-08-01	0.789	1967-01-04
5555	HIROSHIMA	32.987	1995-08-11	6.357	1979-02-15
6666	HIROSHIMA	31.567	1997-09-01	11.258	1990-02-06
7777	HAKATA	37.525	1985-06-30	17.369	1995-12-27

⇩

Results



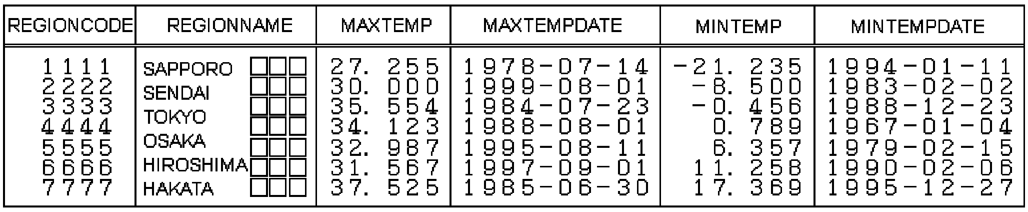
REGIONCODE	MAXTEMPDAY
1111	1978.07.14 FRI
2222	1999.08.01 SUN
3333	1984.07.23 MON
4444	1988.08.01 MON
5555	1995.08.11 FRI
6666	1997.09.01 MON
7777	1985.06.30 SUN

Example 6:

The minimum temperature date in the WEATHERMNTBL table is converted to character-type data in the date format. The date format "DAY MONTH DD, YYYY" is assumed to be specified in CHAR-type host variable :DAYFMT.

```
SELECT REGIONCODE, CNV_CHAR(MINTEMPDATE, :DAYFMT) AS  MINTEMPDAY
  FROM WAETHERMNTBL
```

WEATHERMNTBL table

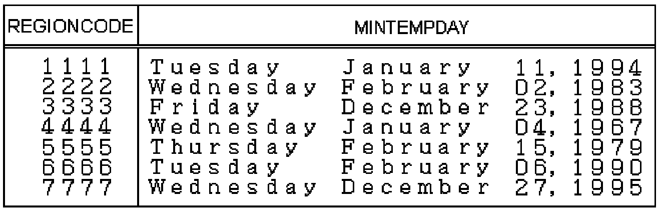


REGIONCODE	REGIONNAME	MAXTEMP	MAXTEMPDATE	MINTEMP	MINTEMPDATE
1111	SAPPORO □□□	27.255	1978-07-14	-21.235	1994-01-11
2222	SENDAI □□□	30.000	1999-08-01	-8.500	1983-02-02
3333	TOKYO □□□	35.554	1984-07-23	-0.456	1988-12-23
4444	OSAKA □□□	34.123	1988-08-01	0.789	1967-01-04
5555	HIROSHIMA□□□	32.987	1995-08-11	6.357	1979-02-15
6666	HIROSHIMA□□□	31.567	1997-09-01	11.258	1990-02-06
7777	HAKATA □□□	37.525	1985-06-30	17.369	1995-12-27

□ : Indicates an em-size null character.

⇩

Results

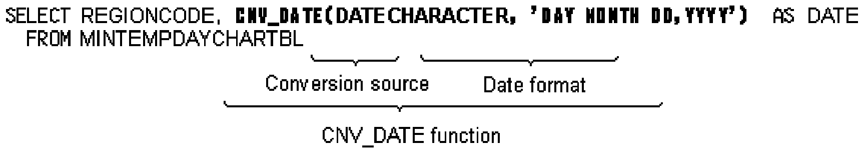


REGIONCODE	MINTEMPDAY
1111	Tuesday January 11, 1994
2222	Wednesday February 02, 1983
3333	Friday December 23, 1988
4444	Wednesday January 04, 1967
5555	Thursday February 15, 1979
6666	Tuesday February 06, 1990
7777	Wednesday December 27, 1995

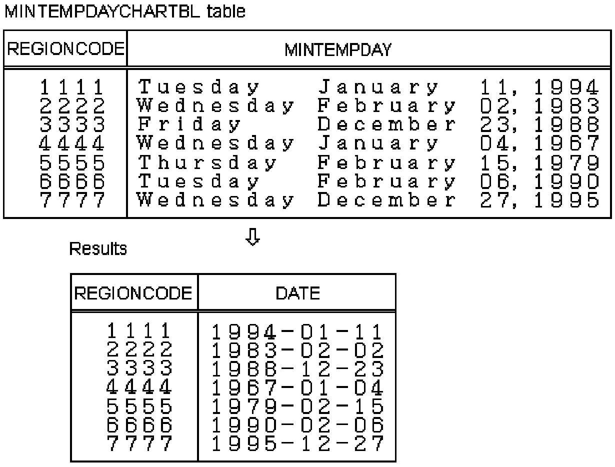
Example 7:

The date characters in MINTEMPDAYCHARTBL are converted to a date.

The character-type data defined in the date format can be converted to date-time-type data using the CNV_DATE function.



```
SELECT REGIONCODE, CNV_DATE(DATECHARACTER, 'DAY MONTH DD,YYYY') AS DATE
  FROM MINTEMPDAYCHARTBL
```



MINTEMPDAYCHARTBL table

REGIONCODE	MINTEMPDAY
1111	Tuesday January 11, 1994
2222	Wednesday February 02, 1983
3333	Friday December 23, 1988
4444	Wednesday January 04, 1967
5555	Thursday February 15, 1979
6666	Tuesday February 06, 1990
7777	Wednesday December 27, 1995

⇩

Results

REGIONCODE	DATE
1111	1994-01-11
2222	1983-02-02
3333	1988-12-23
4444	1967-01-04
5555	1979-02-15
6666	1990-02-06
7777	1995-12-27

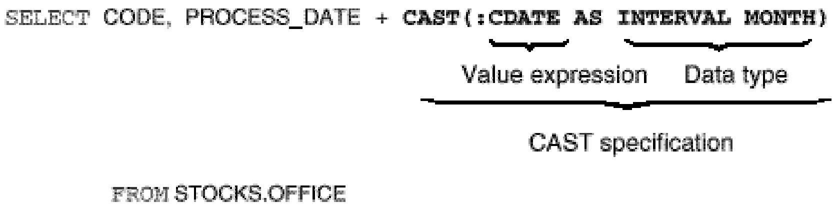
6.6 Converting the Data Type to Manipulate Data

The data types of columns can be converted to execute date or time operations using character strings and to add or update data in time type or time interval type columns.

An example of a date operation executed by converting the character string data type to the time interval type (year and month) is given below.

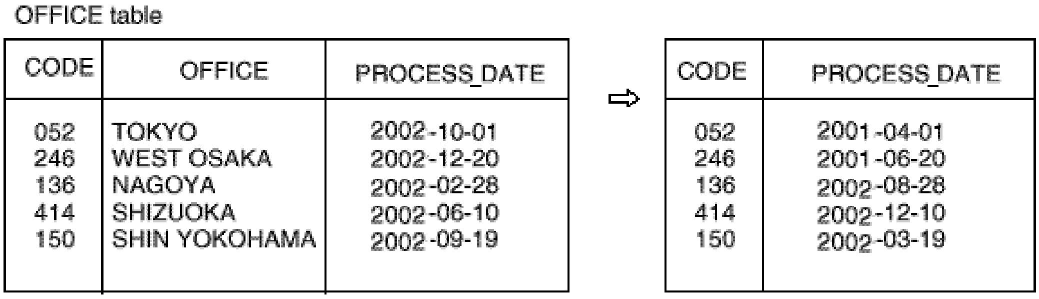
Example 1:

In this example, codes and dates obtained by adding six months to the process date are fetched from the OFFICE table. Also, "6" has been specified for character string type host variable CDATE.



```
SELECT CODE, PROCESS_DATE + CAST(:CDATE AS INTERVAL MONTH)
  FROM STOCKS.OFFICE
```

[Figure: Converting data from character string type to time interval type (year and month)]



OFFICE table

CODE	OFFICE	PROCESS_DATE
052	TOKYO	2002-10-01
246	WEST OSAKA	2002-12-20
136	NAGOYA	2002-02-28
414	SHIZUOKA	2002-06-10
150	SHIN YOKOHAMA	2002-09-19

⇨

CODE	PROCESS_DATE
052	2001-04-01
246	2001-06-20
136	2002-08-28
414	2002-12-10
150	2002-03-19

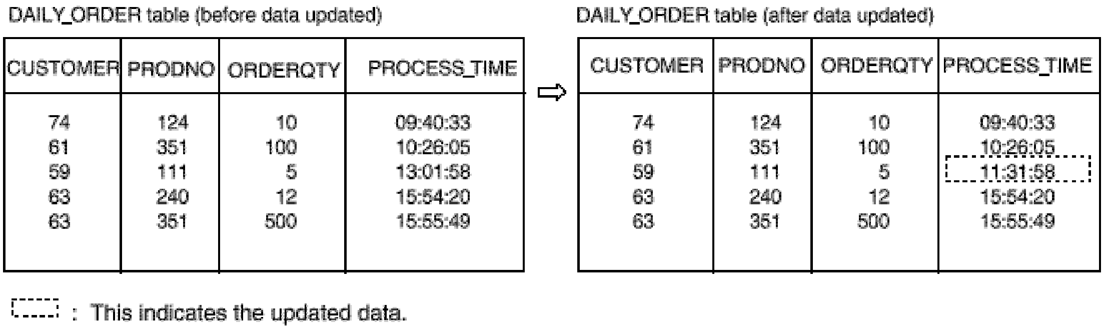
An example of a time operation executed by converting the character string data type to the time interval type (date and time) is given below.

Example 2:

In this example, the process time is updated to a value reduced by 1 hour and 30 minutes for data in the DAILY_ORDER table whose ORDERQTY is less than 10. Also, "1:30" has been specified for character string type host variable OTIME.

```
UPDATE STOCKS.DAILY_ORDER

  SET PROCESS_TIME = PROCESS_TIME - CAST(:OTIME AS INTERVAL HOUR TO MINUTE)

  WHERE ORDERQTY < 10
```

[Figure: Converting data from character string type to time interval type (date and time)]



DAILY_ORDER table (before data updated)

CUSTOMER	PRODNO	ORDERQTY	PROCESS_TIME
74	124	10	09:40:33
61	351	100	10:26:05
59	111	5	13:01:58
63	240	12	15:54:20
63	351	500	15:55:49

⇨

DAILY_ORDER table (after data updated)

CUSTOMER	PRODNO	ORDERQTY	PROCESS_TIME
74	124	10	09:40:33
61	351	100	10:26:05
59	111	5	11:31:58
63	240	12	15:54:20
63	351	500	15:55:49

⬚ : This indicates the updated data.

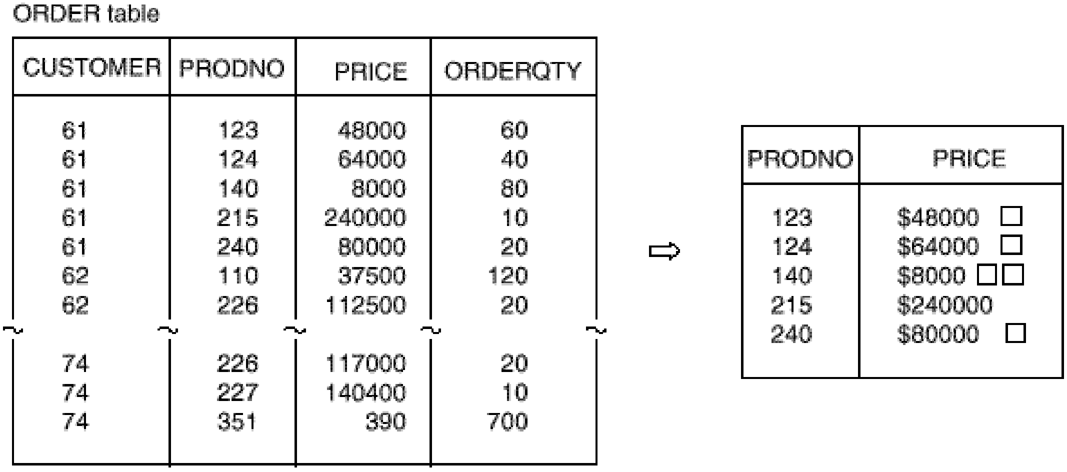
An example of converting the numeric value data type to the character string type and adding to a character string to it is given below.

Example 3:

In this example, PRODNO and PRICE are fetched from the ORDER table for data for CUSTOMER 61, and $ is prefixed to the price.

```
SELECT PRODNO, '$' || CAST(PRICE AS CHARACTER(6))

       FROM STOCKS.ORDER WHERE CUSTOMER = 61
```

[Figure: Converting data from numeric type to character string type]



ORDER table

CUSTOMER	PRODNO	PRICE	ORDERQTY
61	123	48000	60
61	124	64000	40
61	140	8000	80
61	215	240000	10
61	240	80000	20
62	110	37500	120
62	226	112500	20
~	~	~	~
74	226	117000	20
74	227	140400	10
74	351	390	700

⇨

PRODNO	PRICE
123	$48000 ☐
124	$64000 ☐
140	$8000 ☐☐
215	$240000
240	$80000 ☐

☐ : This indicates a null character.

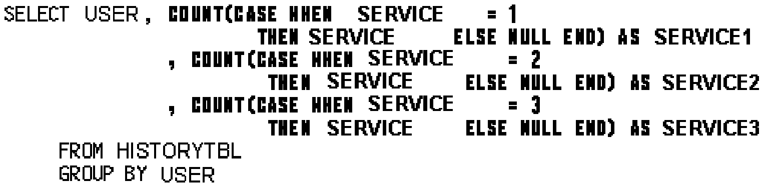
6.7 Using CASE Expression to Manipulate Data

The CASE expression is used to change the value of a results column depending on the column value, arrange fetched columns in a sequence based on priority level, and split data to cases to specify an appropriate value to each case.

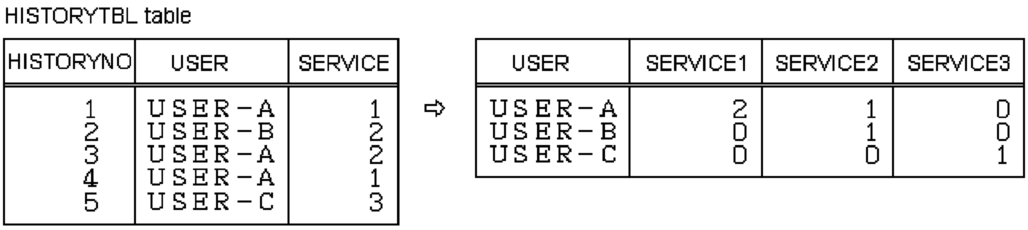
Examples of using the CASE expression are as follows.

Example 1:

The number of services used for each user is obtained from history table HISTORYTBL.



```
SELECT USER, COUNT(CASE WHEN SERVICE   = 1
                   THEN SERVICE    ELSE NULL END) AS SERVICE1
           , COUNT(CASE WHEN SERVICE   = 2
                    THEN SERVICE    ELSE NULL END) AS SERVICE2
           , COUNT(CASE WHEN SERVICE   = 3
                    THEN SERVICE    ELSE NULL END) AS SERVICE3
     FROM HISTORYTBL
     GROUP BY USER
```



HISTORYTBL table

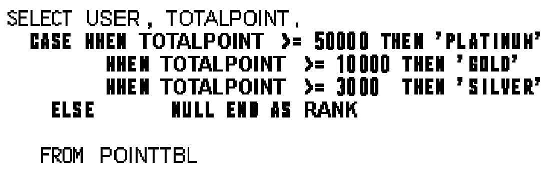
HISTORYNO	USER	SERVICE
1	USER-A	1
2	USER-B	2
3	USER-A	2
4	USER-A	1
5	USER-C	3

⇨

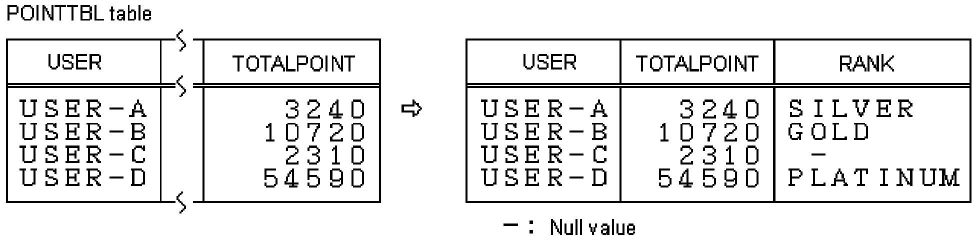
USER	SERVICE1	SERVICE2	SERVICE3
USER-A	2	1	0
USER-B	0	1	0
USER-C	0	0	1

Example 2:

According to the number of points, the users and points indicated by a symbol (PLATINUM, GOLD, or SILVER) are obtained from point table POINTTBL.



```
SELECT USER, TOTALPOINT,
  CASE WHEN TOTALPOINT >= 50000 THEN 'PLATINUM'
       WHEN TOTALPOINT >= 10000 THEN 'GOLD'
       WHEN TOTALPOINT >= 3000  THEN 'SILVER'
   ELSE      NULL END AS RANK

  FROM POINTTBL
```



POINTTBL table

USER	TOTALPOINT
USER-A	3240
USER-B	10720
USER-C	2310
USER-D	54590

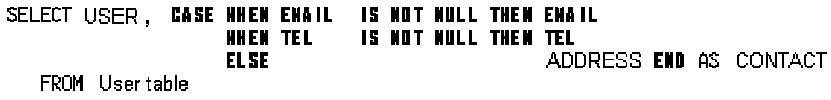
⇨

USER	TOTALPOINT	RANK
USER-A	3240	SILVER
USER-B	10720	GOLD
USER-C	2310	−
USER-D	54590	PLATINUM

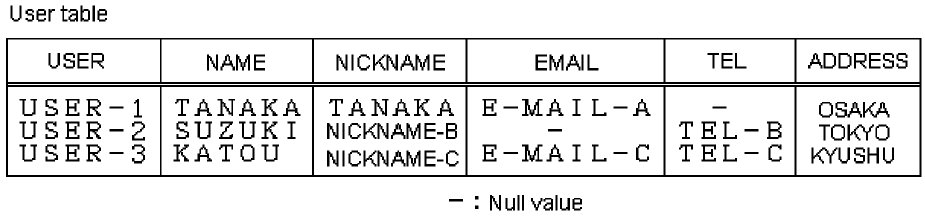
− : Null value

Example 3:

The first setting items are obtained from the user table in the sequence of e-mail address, phone, and address.



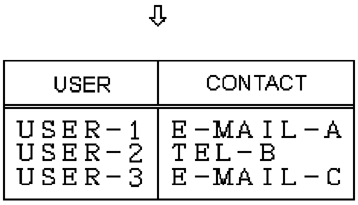
```
SELECT USER, CASE WHEN EMAIL  IS NOT NULL THEN EMAIL
                  WHEN TEL    IS NOT NULL THEN TEL
                  ELSE                         ADDRESS END AS CONTACT
  FROM User table
```



User table

USER	NAME	NICKNAME	EMAIL	TEL	ADDRESS
USER-1	TANAKA	TANAKA	E-MAIL-A	−	OSAKA
USER-2	SUZUKI	NICKNAME-B	−	TEL-B	TOKYO
USER-3	KATOU	NICKNAME-C	E-MAIL-C	TEL-C	KYUSHU

− : Null value



⇩

USER	CONTACT
USER-1	E-MAIL-A
USER-2	TEL-B
USER-3	E-MAIL-C

Example 4:

The name of users and points of the users who reach the next rank by 500 points are obtained from the POINTTBL table. The number of points of each rank are as follows: SILVER is 3,000 points; GOLD is 10,000 points; and PLATINUM is 50,000 points.

```
SELECT USER, TOTAL POINT  FROM  POINTTBL
  WHERE  POINT  >= CASE WHEN  POINT  <= 3000 THEN 2500
                        WHEN  POINT  <=10000 THEN 9500
                        WHEN  POINT  <=50000 THEN 49500 END
```

Point table

USER	TOTALPOINT
USER-A	2780
USER-B	10720
USER-C	2310
USER-D	54590
USER-E	9830

⇨

USER	TOTALPOINT
USER-A	2780
USER-E	9830

Example 5:

The services used are grouped, and the total number of services used for each group is obtained, from history table HISTORYTBL.

```
SELECT  SERVICEGRP, COUNT(*)  AS  TOTAL
 FROM HISTORYTBL
 GROUP BY  CASE WHEN SERVICE    IN (1,4,7) THEN 'SERVICEGRP1'
                WHEN SERVICE    IN (2,5,8) THEN 'SERVICEGRP2'
                ELSE 'SERVICEGRP3'   END  AS SERVICEGRP
```

History table

HISTORYNO	USER	SERVICE
1	USER-A	1
2	USER-B	2
3	USER-A	5
4	USER-A	7
5	USER-C	3

⇨

SERVICEGRP	TOTAL
SERVICEGRP1	2
SERVICEGRP2	2
SERVICEGRP3	1

Example 6:

The service symbols listed in history table HISTORYTBL are changed.

```
UPDATE HISTORYTBL
  SET SERVICE =   CASE WHEN SERVICE  =  1  THEN  111
                       WHEN SERVICE  =  2  THEN  222
                       WHEN SERVICE  =  3  THEN  333
                       ELSE 999                    END
```

History table

HISTORYNO	USER	SERVICE
1	USER-A	1
2	USER-B	2
3	USER-A	5
4	USER-A	7
5	USER-C	3

⇨

HISTORYNO	USER	SERVICE
1	USER-A	111
2	USER-B	222
3	USER-A	999
4	USER-A	999
5	USER-C	333

6.8 Specifying Row IDs to Manipulate Data

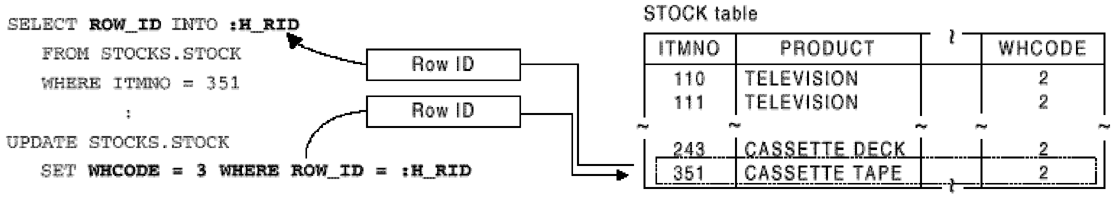
A row ID (ROW_ID) recognizes a unique row of a table in the database. Use a single row SELECT statement or FETCH statement to fetch a row ID. The host variable for receiving the row ID must correspond to the ROW_ID. For the variable definition corresponding to each data type, see Table "SQL data types and equivalent C language host variable data types" and Table "SQL date types and equivalent data types for COBOL host variables". Specify the fetched row ID in the following search conditions to manipulate the row data.

- Search condition of the UPDATE statement:search
- Search condition of the DELETE statement:search
- Search condition of the WHERE clause in a query specification

The following shows an example of specifying row IDs.

Example:

In this example, data is updated by using row IDs.



```
SELECT ROW_ID INTO :H_RID
   FROM STOCKS.STOCK
   WHERE ITMNO = 351
          :
UPDATE STOCKS.STOCK
   SET WHCODE = 3 WHERE ROW_ID = :H_RID
```

STOCK table

ITMNO	PRODUCT		WHCODE
110	TELEVISION		2
111	TELEVISION		2
243	CASSETTE DECK		2
351	CASSETTE TAPE		2

6.9 Manipulating Data using Parallel Query

If a key item in the data search is known for jobs such as ordinary jobs, the efficiency of the data search can be improved by preparing an index or similar facility. If the database for an ordinary job is used for document output in batch processing at night, the search time may increase. To improve the efficiency of the batch processing, divide the database into DSIs and perform a parallel search of the data in each DSI. This processing is called parallel query. To use parallel query, make the following specifications:

- Specify PARALLEL for parallel query in the cursor declaration.
- Specify SQL96 for SQL_LEVEL in the application environment file.
- Specify the maximum number of parallel instances in MAX_PARALLEL in the application environment file.

For the division of a database into multiple DSIs, refer to the "RDB User's Guide: Database Definition." For application environment files, refer to the "RDB User's Guide: Application Program Development."

The following is an example of specifying a parallel query.

Example:

Determine the sales amount for each product code by year and month.

Application

```
DECLARE CSR1 CURSOR FOR
   SELECT  SELECT PRODCODE, YEAR&MONTH, SUM (SALES_AMOUNT) FROM SALES_HISTORY
     WHERE PROD_CATEGORY IN (N 'ELEC_APPLIANCES') GROUP BY PRODCODE, YEAR&MONTH PARALLEL
          ⋮
OPEN   CSR1
          ⋮
FETCH  CSR1 INTO :H2, H3
          ⋮
```

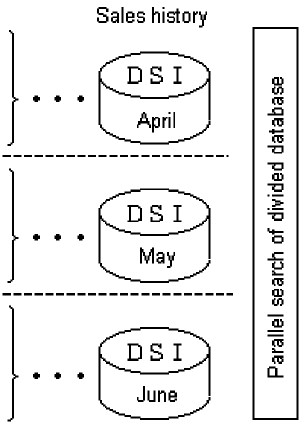
Parallel query specification

Application environment file

```
SQL_LEVEL = SQL96
MAX_PARALLEL = 3
```

Sales history

RODCODE	PRODUCT	PROD_CATEGORY	YEAR& MONTH	WEEK	SALES_AMOUNT
2005	DICTIONARIES	BOOKS	9804	1	52000
3104	STUFFED TOYS	TOYS	9804	1	106000
1000	AIR-CONDITIONERS	ELEC_APPLIANCES	9804	1	205000
1020	WASHING_MACHINES	ELEC_APPLIANCES	9804	1	210000
1000	AIR-CONDITIONERS	ELEC_APPLIANCES	9804	3	180000
1020	WASHING_MACHINES	ELEC_APPLIANCES	9804	3	258000
2005	DICTIONARIES	BOOKS	9805	1	30000
3104	STUFFED TOYS	TOYS	9805	1	93200
1000	AIR-CONDITIONERS	ELEC_APPLIANCES	9805	1	350000
1020	WASHING_MACHINES	ELEC_APPLIANCES	9805	1	395000
1000	AIR-CONDITIONERS	ELEC_APPLIANCES	9805	3	410000
1020	WASHING_MACHINES	ELEC_APPLIANCES	9805	3	420000
2005	DICTIONARIES	BOOKS	9806	1	42000
3104	STUFFED TOYS	TOYS	9806	1	89000
1000	AIR-CONDITIONERS	ELEC_APPLIANCES	9806	1	370000
1020	WASHING_MACHINES	ELEC_APPLIANCES	9806	1	430000
1000	AIR-CONDITIONERS	ELEC_APPLIANCES	9806	3	525000
1020	WASHING_MACHINES	ELEC_APPLIANCES	9806	3	495000



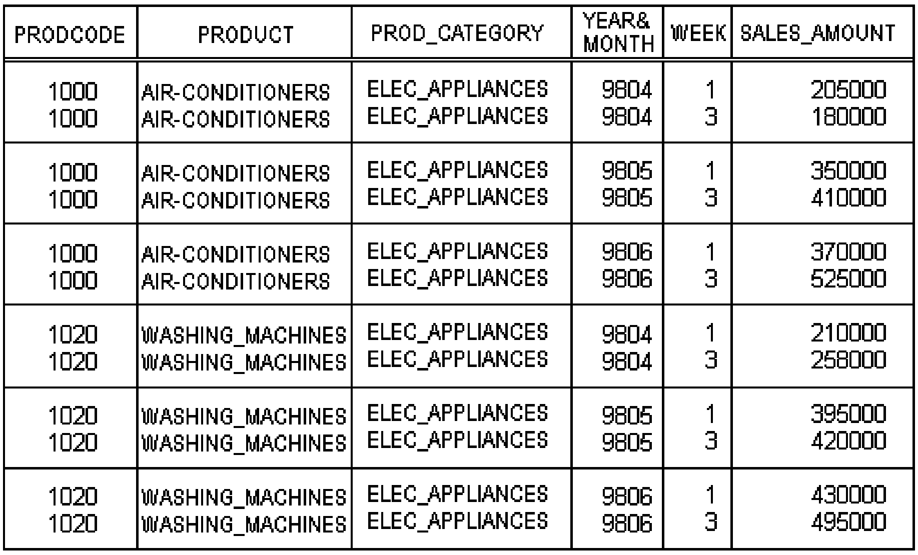
⇩

Table derived from WHERE clause

RODCODE	PRODUCT	PROD_CATEGORY	YEAR& MONTH	WEEK	SALES_AMOUNT
1000	AIR-CONDITIONERS	ELEC_APPLIANCES	9804	1	205000
1020	WASHING_MACHINES	ELEC_APPLIANCES	9804	1	210000
1000	AIR-CONDITIONERS	ELEC_APPLIANCES	9804	3	180000
1020	WASHING_MACHINES	ELEC_APPLIANCES	9804	3	258000
1000	AIR-CONDITIONERS	ELEC_APPLIANCES	9805	1	350000
1020	WASHING_MACHINES	ELEC_APPLIANCES	9805	1	395000
1000	AIR-CONDITIONERS	ELEC_APPLIANCES	9805	3	410000
1020	WASHING_MACHINES	ELEC_APPLIANCES	9805	3	420000
1000	AIR-CONDITIONERS	ELEC_APPLIANCES	9806	1	370000
1020	WASHING_MACHINES	ELEC_APPLIANCES	9806	1	430000
1000	AIR-CONDITIONERS	ELEC_APPLIANCES	9806	3	525000
1020	WASHING_MACHINES	ELEC_APPLIANCES	9806	3	495000

⇩

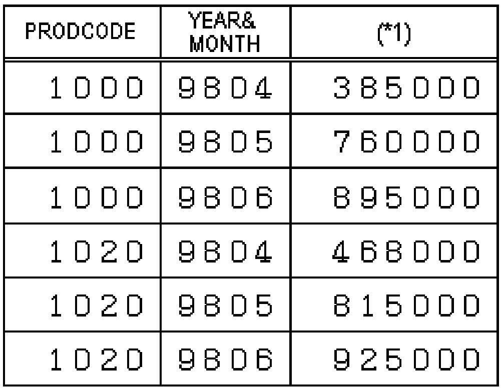
Table derived from GROUP BY clause



PRODCODE	PRODUCT	PROD_CATEGORY	YEAR& MONTH	WEEK	SALES_AMOUNT
1000	AIR-CONDITIONERS	ELEC_APPLIANCES	9804	1	205000
1000	AIR-CONDITIONERS	ELEC_APPLIANCES	9804	3	180000
1000	AIR-CONDITIONERS	ELEC_APPLIANCES	9805	1	350000
1000	AIR-CONDITIONERS	ELEC_APPLIANCES	9805	3	410000
1000	AIR-CONDITIONERS	ELEC_APPLIANCES	9806	1	370000
1000	AIR-CONDITIONERS	ELEC_APPLIANCES	9806	3	525000
1020	WASHING_MACHINES	ELEC_APPLIANCES	9804	1	210000
1020	WASHING_MACHINES	ELEC_APPLIANCES	9804	3	258000
1020	WASHING_MACHINES	ELEC_APPLIANCES	9805	1	395000
1020	WASHING_MACHINES	ELEC_APPLIANCES	9805	3	420000
1020	WASHING_MACHINES	ELEC_APPLIANCES	9806	1	430000
1020	WASHING_MACHINES	ELEC_APPLIANCES	9806	3	495000

⇩

Table derived from query specification



PRODCODE	YEAR& MONTH	(*1)
1000	9804	385000
1000	9805	760000
1000	9806	895000
1020	9804	468000
1020	9805	815000
1020	9806	925000

*1: The result column of the group function has no name.

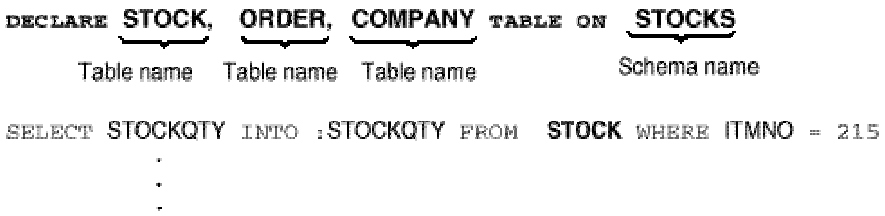
6.10 Omitting Schema Names

In the examples of SQL statements provided so far, table names have been qualified with a schema name, such as in STOCKS.STOCK. This section explains how to use a table declaration in which the schema name qualifier is not specified.

Specify the table name used by the application program and the name of the schema in which the table is located in the table declaration. After the table declaration has been specified, when an SQL statement is entered, the schema name qualifier can be omitted from the table name specification. The following is an example of specifying a table declaration:

Example:

In this example, the STOCK table, the ORDER table, and the COMPANY table have been declared in the table declaration so the schema name qualifier can be omitted from the table name specification.



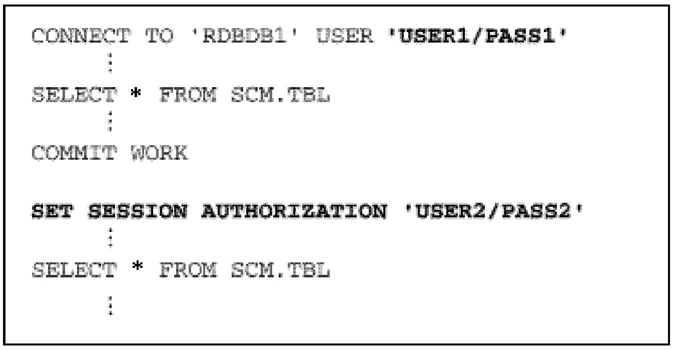
```
DECLARE STOCK, ORDER, COMPANY TABLE ON STOCKS

SELECT STOCKQTY INTO :STOCKQTY FROM STOCK WHERE ITMNO = 215
           .
           .
           .
```

6.11 Changing the User of the Current Session

When an application program accesses a database, the user of the current session becomes the person connected to the database. This user can be changed in the application program.

Example:

In this example, the user of the current session is changed to USER2.



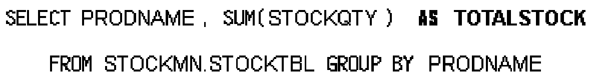
```
CONNECT TO 'RDBDB1' USER 'USER1/PASS1'
      ⋮
SELECT * FROM SCM.TBL
      ⋮
COMMIT WORK

SET SESSION AUTHORIZATION 'USER2/PASS2'
      ⋮
SELECT * FROM SCM.TBL
      ⋮
```

6.12 Adding Name to Results Column in Query Specification

If a function is specified in the select column list of the query specification, no name is added to the derived columns. To add a name, use the AS clause and add a column name. An example of specification is shown below.

Example:

The total stock quantity for each product name is fetched from stock table STOCKTBL. In this example, use the table derived from the query specification to add a name to the total stock quantity column.



```
SELECT PRODNAME , SUM(STOCKQTY ) AS TOTALSTOCK
   FROM STOCKMN.STOCKTBL GROUP BY PRODNAME
```

Table derived from GROUP BY clause

PRODNO	PRODNAME	STOCKQTY	WAREHOUSENO	
110	TELEVISION	85	2	One group
111	TELEVISION	90	2	
212	TELEVISION	0	2	
215	VIDEO	5	2	One group
123	REFRIGERATOR	60	1	One group
124	REFRIGERATOR	75	1	
226	REFRIGERATOR	8	1	
227	REFRIGERATOR	15	1	
140	CASSETTE DECK	120	2	One group
240	CASSETTE DECK	25	2	
243	CASSETTE DECK	14	2	
351	CASSETTE TAPE	2500	2	One group

⇩

Table derived from query specification

PRODNAME	TOTALSTOCK (*1)
TELEVISION	175
VIDEO	5
REFRIGERATOR	158
CASSETTE DECK	159
CASSETTE TAPE	2500

*1 The name is added to the results column of the set function.

6.13 Manipulating Data Using Sequence

CURRVAL and NEXTVAL can be used to fetch the created sequence number. CURRVAL fetches the current sequence number in the application program. NEXTVAL fetches the next value after the last sequence number fetched in the RDB system.

CURRVAL and NEXTVAL can be specified in the following parts of SQL statements in the application program or stored procedure:

- Select column list of a single row SELECT statement
- Select column list for a query specification (query expression for cursor specification only)
- SET clause of the UPDATE statement
- Select column list for a query specification specified in the INSERT statement
- VALUES clause of the INSERT statement
- SET statement of a stored procedure
- Triggered SQL statement

CURRVAL and NEXTVAL cannot be specified in the following parts of SQL statements:

- Search condition of a WHERE clause
- Select column list for a subquery
- DEFAULT clause of ALTER TABLE ADD COLUMN

For the SELECT statement, sequence numbers are assigned to the rows of a search result.

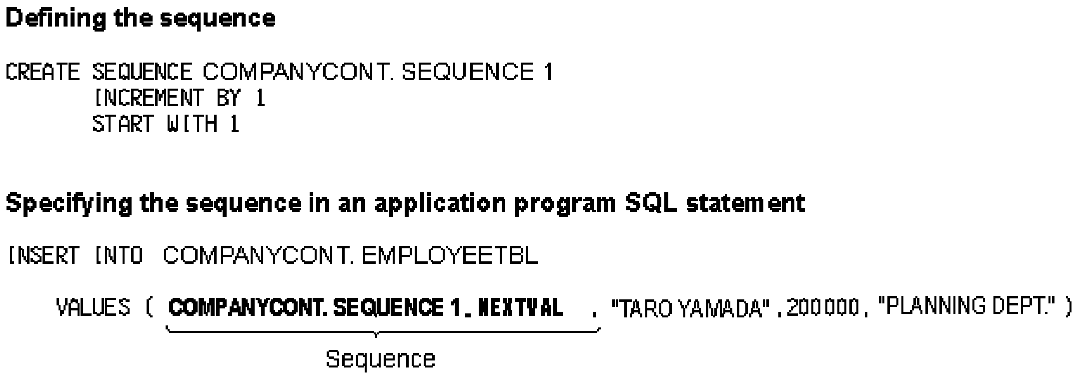
For the UPDATE statement, sequence numbers are assigned to update rows. For the INSERT statement, sequence numbers are assigned to the insertion rows of the insertion destination. In either case, if there are several search results, sequence numbers are assigned to each row of the results.

To fetch the first sequence in a session, use NEXTVAL.

If several NEXTVALs of the same sequence are specified in one SQL statement, the same sequence number is returned for all the NEXTVALs. If both the NEXTVAL and CURRVAL of the same sequence are specified in one SQL statement, the same sequence number is returned for both without reference to the order in which they appear in the SQL statement.

Example 1:

This example shows the method for specifying a sequence in the SQL statement of an application program. In the example, employee numbers (EMPLOYEENO) are incremented and assigned to new employees.



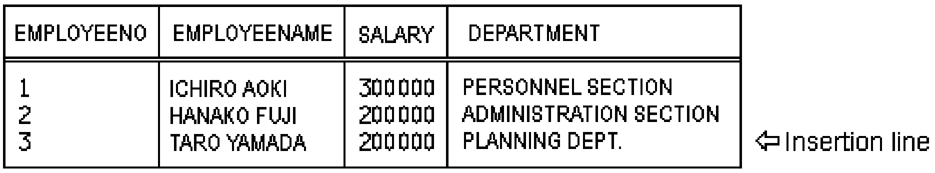
Defining the sequence

```
CREATE SEQUENCE COMPANYCONT. SEQUENCE 1
       INCREMENT BY 1
       START WITH 1
```

Specifying the sequence in an application program SQL statement

```
INSERT INTO  COMPANYCONT. EMPLOYEETBL

    VALUES ( COMPANYCONT. SEQUENCE 1. NEXTVAL  , "TARO YAMADA" , 200000, "PLANNING DEPT." )
             └──────────────────┬───────────────┘
                             Sequence
```

EMPLOYEETBL table



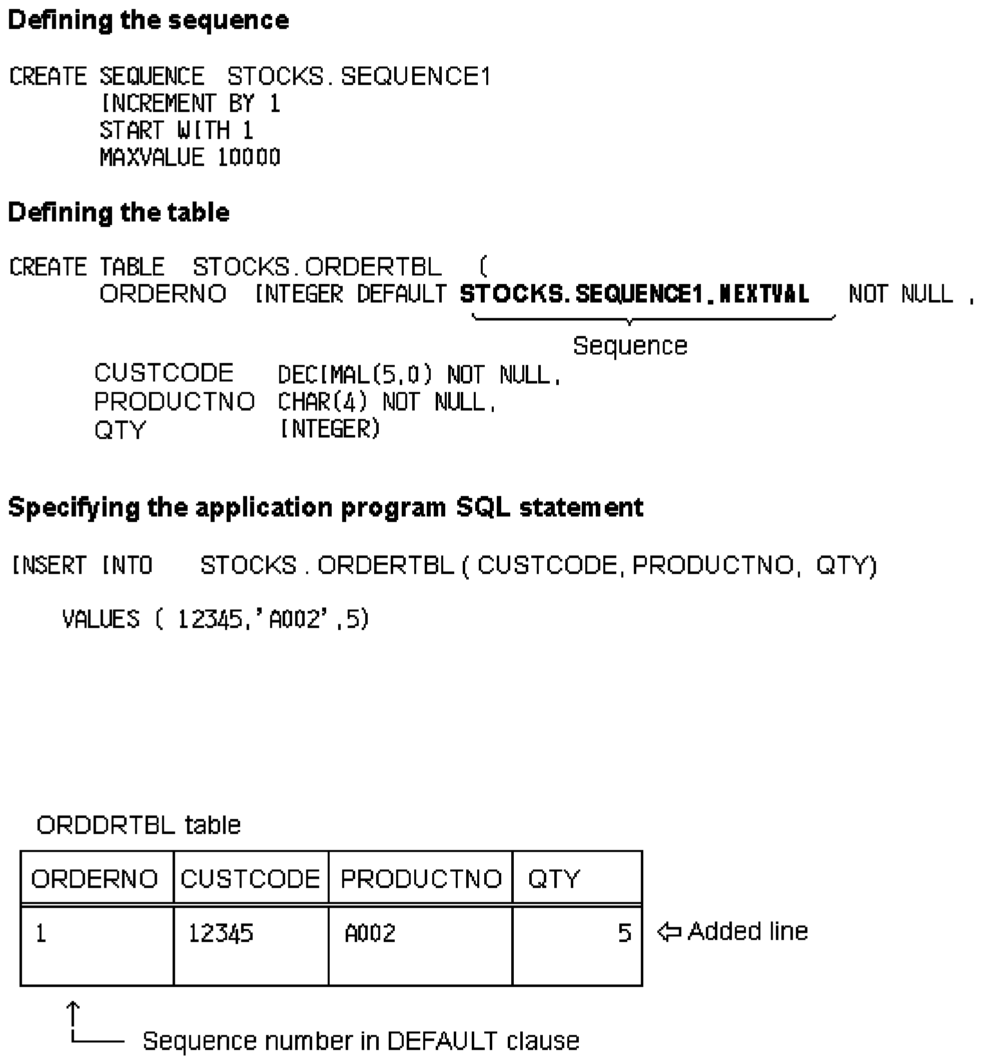
EMPLOYEENO	EMPLOYEENAME	SALARY	DEPARTMENT
1	ICHIRO AOKI	300000	PERSONNEL SECTION
2	HANAKO FUJI	200000	ADMINISTRATION SECTION
3	TARO YAMADA	200000	PLANNING DEPT.

⇦ Insertion line

Example 2:

This example shows the method for automatically inserting a sequence number in an insertion line using a sequence. In the example, new order numbers (ORDERNO) are assigned by incrementing order numbers (ORDERNO) in the order table (ORDERTBL).

To insert a sequence number automatically into an insertion line, use the DEFAULT clause to specify a sequence in a table definition column. In the INSERT statement of the application program, omit the column where a sequence is specified in the DEFAULT clause or specify the DEFAULT clause as the insertion value.



Defining the sequence

```
CREATE SEQUENCE  STOCKS.SEQUENCE1
       INCREMENT BY 1
       START WITH 1
       MAXVALUE 10000
```

Defining the table

```
CREATE TABLE  STOCKS.ORDERTBL  (
       ORDERNO  INTEGER DEFAULT STOCKS.SEQUENCE1.NEXTVAL  NOT NULL ,
                                 \________________________/
                                          Sequence
       CUSTCODE   DECIMAL(5,0) NOT NULL,
       PRODUCTNO  CHAR(4) NOT NULL,
       QTY        INTEGER)
```

Specifying the application program SQL statement

```
INSERT INTO    STOCKS . ORDERTBL ( CUSTCODE, PRODUCTNO,  QTY)

    VALUES ( 12345,'A002',5)
```

ORDDRTBL table

ORDERNO	CUSTCODE	PRODUCTNO	QTY
1	12345	A002	5

⇦ Added line

↑ Sequence number in DEFAULT clause

Example 3:

This example shows the method for referencing the sequence numbers for stock control (STOCKCTL) and sequence 1 (SEQ-1) by specifying CURRVAL and NEXTVAL.

Reference the current sequence number by specifying CURRVAL.

```
SELECT STOCKS.SEQUENCE1.CURRVAL
                 Sequence
       FROM RDB11_SYSTEM.RDB11_ASSISTTABLE   (*1)
```

Reference the next sequence number by specifying NEXTVAL.

```
SELECT STOCKS.SEQUENCE1.NEXTVAL
                 Sequence
       FROM RDB11_SYSTEM.RDB11_ASSISTTABLE    (*1)
```

*1 RDB11_ASSISTTABLE is a table that simplifies the execution of SQL statements.
This table contains one data item with value 1.
The table can be used when the current time and CURRVAL value are obtained by executing a single row SELECT statement specifying the date/time function and sequence in the select column list.

Chapter 7 Executing Dynamic SQL Statements When Application Programs are Executed

This chapter describes the functions and uses of dynamic SQL.

This chapter consists of the following sections:

7.1 Overview of Dynamic SQL

7.2 Dynamically Modifying and Executing SQL Statements

7.3 Dynamically Modifying and Executing SQL Statement Conditions

7.4 Immediately Executing SQL Statements

7.5 Changing Database Names and Omitted Schema Names

7.1 Overview of Dynamic SQL

Dynamic SQL is used to generate and execute SQL statements when application programs are executed. For example, suppose an application program to delete rows from a table is to be developed using delete row search conditions entered from an external data set. In this case, SQL statements to delete rows cannot be specified since the search condition to delete rows cannot be determined when the program is coded. SQL statements cannot be completed until the search condition is read from the external data set when the application program is executed. Embedded SQL statements cannot be used to write this kind of application program since SQL statements must be completed at compilation because the embedded SQL statements are interpreted at compilation. Use dynamic SQL for this kind of application program. With dynamic SQL, the SQL statements are interpreted and executed at application program run-time.

This section explains the following elements required to execute dynamic SQL:

- Overview of Dynamic SQL
- Preparable statements
- Dynamic parameter specification
- USING clause

■SQLDA structure and SQL descriptor area

If SQL statements are dynamically modified to manipulate data (i.e., if an SQL statement is entered outside an application program), the SQL statements to be entered are unknown when an application program is created. When data is fetched from a database, for example, the variable used to fetch the data must be declared in the application program. However, if the SQL statement to be entered is not known, a variable cannot be declared. This situation is the same as when data is entered in the database. Therefore, when modifying SQL statements dynamically, allocate a temporary descriptor area for data manipulation. When this descriptor area is used, data can be manipulated simply by declaring a proper variable.

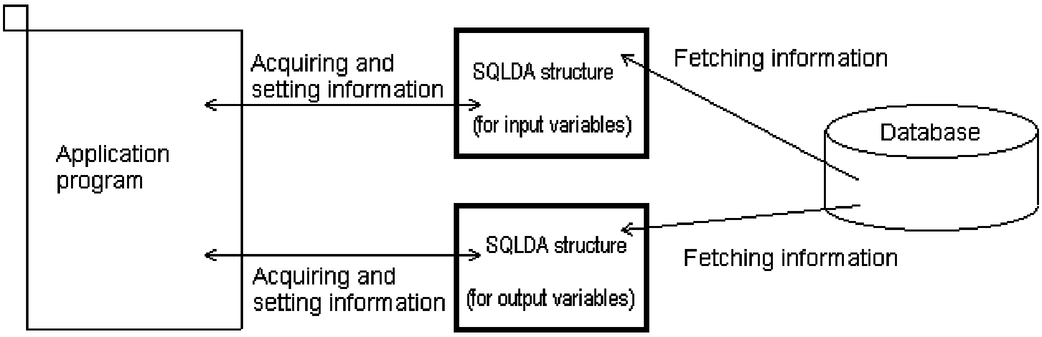
The descriptor area is used as an SQLDA structure or an SQL descriptor area. The SQLDA structure can be used only in C.

◆SQLDA structure

By dynamically allocating and using an SQLDA structure area in an application program, the user can directly manipulate SQLDA structure information.

When the SQLDA structure is used, processing is simplified only if the dynamic parameter specification and select column list information are set in an SQLDA structure member or are obtained from a member.

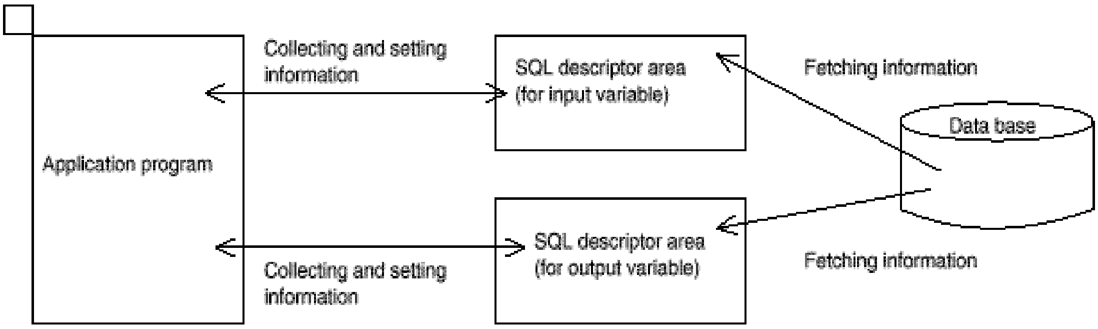
The SQLDA structure is positioned as shown in the figure below.


Application program
Acquiring and setting information
SQLDA structure (for input variables)
Fetching information
Database
Acquiring and setting information
SQLDA structure (for output variables)
Fetching information

■SQL descriptor area

An SQL statement may be dynamically modified to manipulate data. In other words, the SQL statement may be entered from outside the application program. In this case, however, the type of SQL statement to be entered cannot be determined when the application program is created. For example, a variable used to fetch data from a data base must normally be declared in the application program. However, if the type of SQL statements to be entered is not known, the variable cannot be declared. This is also the case when data is to be inserted into a data base.

So, the system allocates a temporary work area, the SQL descriptor area, to manipulate data to dynamically modify SQL statements. After declaring appropriate variables in the application program, use this work area to manipulate data. The location of the SQL descriptor area is as follows.


Application program
Collecting and setting information
SQL descriptor area (for input variable)
Fetching information
Data base
Collecting and setting information
SQL descriptor area (for output variable)
Fetching information

■Preparable statements

Certain SQL statements can be dynamically created and executed at run-time after preparation. These statements are:

- Single row SELECT statement
- INSERT statement
- UPDATE statement (searched)
- DELETE statement (searched)
- Multiple row SELECT statement
- UPDATE statement (positioned)
- DELETE statement (positioned)
- Schema definition statement
- Schema manipulation statement
- Storage structure definition statement
- Storage structure manipulation statement
- User control statement
- Access control statement

Of these SQL statements, the following can only be used in this way: single row SELECT statement, multiple row SELECT statement, UPDATE statement (positioned), and DELETE statement (positioned). The syntax of the single row SELECT statement is the same as that of the query specification.

The SET USER PASSWORD statement (user control statement) and SET ROLE statement (access control statement) cannot be executed as preparable statements.

The syntax of the multiple row SELECT statement is the same as that of the cursor specification.

To execute these statements, the statements must be prepared for execution. These statements then become prepared statements.

The following are examples of preparable statements:

Example 1:

In this example, a single row SELECT statement is specified. This statement targets rows to be fetched in which ITMNO is "110" in a table containing columns STOCKQTY and WHCODE derived from the STOCK table.

```
SELECT STOCKQTY,  WHCODE FROM STOCKS. STOCK WHERE ITMNO = 110
```

Example 2:

In this example, a single row SELECT statement is specified. This statement fetches values in ascending order from column STOCKQTY in a table containing columns PRODUCT and STOCKQTY derived from the STOCK table.

```
SELECT PRODUCT,  STOCKQTY  FROM  STOCKS. STOCK ORDER BY STOCKQTY ASC
```

Example 3:

In this example, an UPDATE statement (positioned) is specified. This statement changes PRODUCT to "BATTERY" and WHCODE to "5" in the row in which cursor CU1 is positioned.

```
UPDATE STOCKS. STOCK

       SET  PRODUCT = 'BATTERY,'  WHCODE = 5 WHERE CURRENT OF CU1
```

■Dynamic parameter specification

To specify a value specification (input variable) in a preparable statement, enter a question mark "?" in the value specification section. The question mark is known as the dynamic parameter specification. The dynamic parameter specification is equivalent to the variable specification in an embedded SQL statement. Dynamic parameter specifications can be specified in the following locations in preparable statements:

- In the search condition of a dynamic SELECT statement, single row SELECT statement, UPDATE statement (searched), or DELETE statement (searched)
- In the set clause of an UPDATE statement (searched) or dynamic preferable UPDATE statement (searched)
- In the insert value list of an INSERT statement

The following are examples of dynamic parameter specifications:

Example 4:

In this example, the dynamic parameter specification is specified in the search condition of a DELETE statement (searched).

```
DELETE FROM STOCKS. STOCK WHERE PRODUCT = ?
```

Example 5:

In this example, the dynamic parameter specification is specified in the set clause of an UPDATE statement (searched).

```
UPDATE  STOCKS. STOCK
```

```
          SET  STOCKQTY = 0, WHCODE = ?  WHERE  ITMNO = 111
```

Example 6:

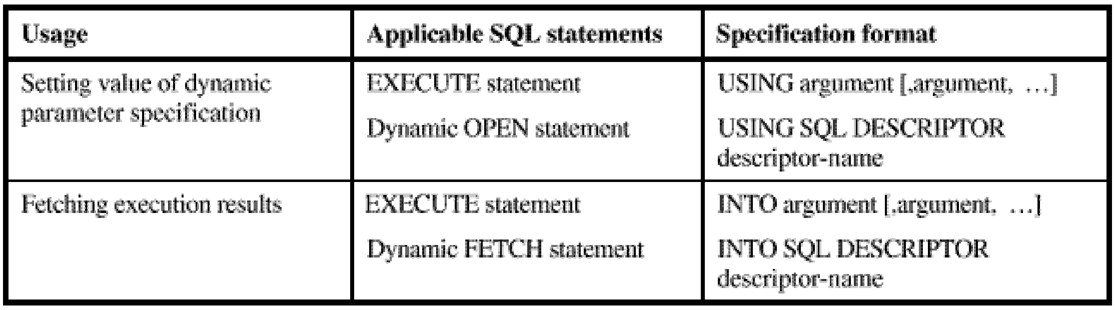
In this example, the dynamic parameter specification is specified in the insert value list of an INSERT statement.

```
   INSERT INTO STOCKS. STOCK  (ITMNO,  PRODUCT,  STOCKQTY)  VALUES(?, ?, ?)
```

■USING clause

Use the USING clause to set values in the dynamic parameter specification of a prepared statement and to fetch the execution results. Table: Uses of the USING clause and its specification format lists the uses of the USING clause and its specification format.

[Table: Uses of the USING clause and its specification format]



Usage	Applicable SQL statements	Specification format
Setting value of dynamic parameter specification	EXECUTE statement Dynamic OPEN statement	USING argument [,argument, …] USING SQL DESCRIPTOR descriptor-name
Fetching execution results	EXECUTE statement Dynamic FETCH statement	INTO argument [,argument, …] INTO SQL DESCRIPTOR descriptor-name

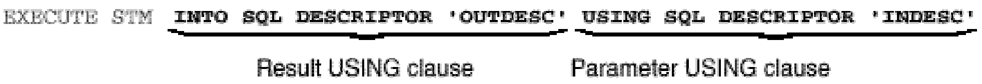
With reference to the EXECUTE statements in Table: Uses of the USING clause and its specification format, the USING clause used to set the value of the dynamic parameter specification is known as the parameter USING clause. The USING clause used to fetch the execution results is known as the result USING clause.

There are two USING clause specification formats:

- Argument specification
- Specification of a variable name or the SQL descriptor name of an SQLDA structure variable

The first is referred to as a USING variable and the second as a USING descriptor.

The method for specifying the USING descriptor in the EXECUTE statement is shown below.



```
EXECUTE STM INTO SQL DESCRIPTOR 'OUTDESC' USING SQL DESCRIPTOR 'INDESC'
```

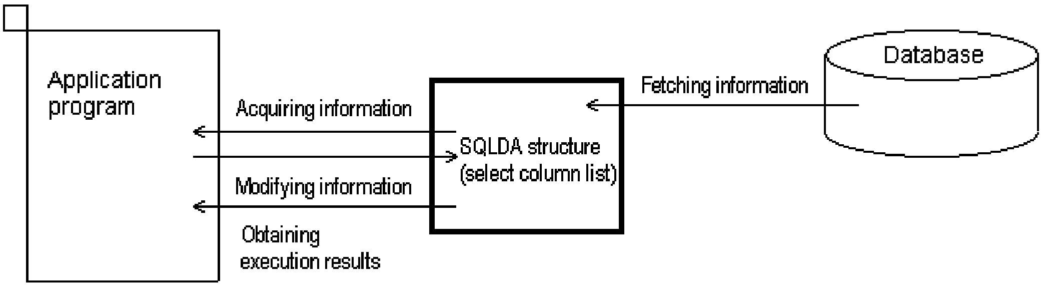
7.2 Modifying and Executing SQL Statements Dynamically

This section explains how to modify and execute SQL statements dynamically.

- Preparing SQL statements and manipulating the SQLDA structure
    - Preparing and executing a dynamic SELECT statement (for SQLDA structure)
    - Preparing and executing a dynamic single row SELECT statement (for SQLDA structure)
- Preparing SQL statements and manipulating the SQL descriptor area
    - Preparing and executing a dynamic SELECT statement (for the SQL descriptor area)
    - Preparing and executing a dynamic single row SELECT statement (for the SQL descriptor area)

7.2.1 Preparing SQL statements and manipulating the SQLDA structure

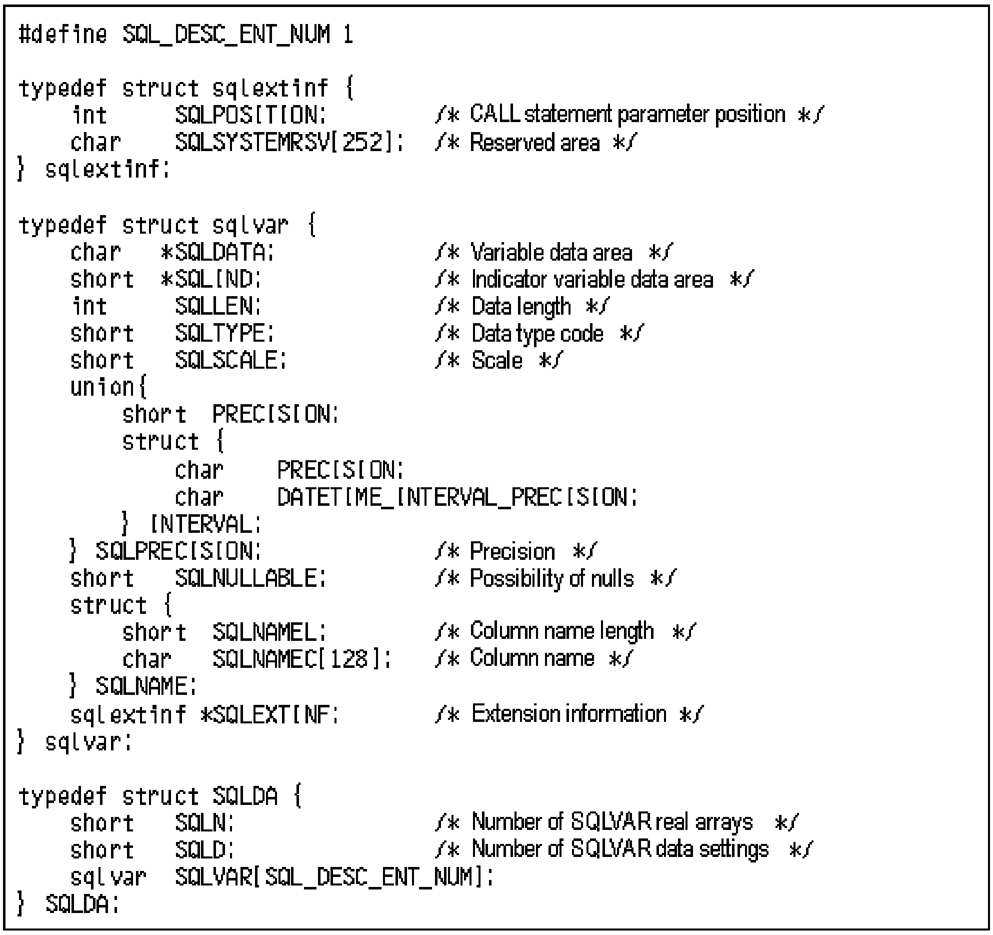
When data is fetched from a database using dynamic SQL, specify the target that will receive the execution results. For the specified target to be able to receive the execution results, the select column list information must match the attributes of the specified target. Matching is accomplished by fetching the select column list information from the database to the SQLDA structure, then modifying the attributes. The SQLDA structure is positioned as shown in the figure below.


Application program
Acquiring information
Modifying information
Obtaining execution results
SQLDA structure (select column list)
Fetching information
Database

The SQLDA structure provides information about the target specification and dynamic parameter specifications.

The SQLDA structure consists of SQLVAR (item descriptor area), SQLN (maximum number of SQLVAR elements), and SQLD (number of valid elements).

The format of the SQLDA structure is as follows.



```
#define SQL_DESC_ENT_NUM 1

typedef struct sqlextinf {
    int     SQLPOSITION;        /* CALL statement parameter position */
    char    SQLSYSTEMRSV[252];  /* Reserved area */
} sqlextinf;

typedef struct sqlvar {
    char   *SQLDATA;            /* Variable data area */
    short  *SQLIND;             /* Indicator variable data area */
    int     SQLLEN;             /* Data length */
    short   SQLTYPE;            /* Data type code */
    short   SQLSCALE;           /* Scale */
    union{
        short  PRECISION;
        struct {
            char    PRECISION;
            char    DATETIME_INTERVAL_PRECISION;
        } INTERVAL;
    } SQLPRECISION;             /* Precision */
    short   SQLNULLABLE;        /* Possibility of nulls */
    struct {
        short  SQLNAMEL;        /* Column name length */
        char   SQLNAMEC[128];   /* Column name */
    } SQLNAME;
    sqlextinf *SQLEXTINF;       /* Extension information */
} sqlvar;

typedef struct SQLDA {
    short   SQLN;               /* Number of SQLVAR real arrays */
    short   SQLD;               /* Number of SQLVAR data settings */
    sqlvar  SQLVAR[SQL_DESC_ENT_NUM];
} SQLDA;
```

The configuration element members of the SQLDA structure have the following meaning:

■SQLN

SQLN indicates the maximum number of the dynamic parameter specifications that can be written or the maximum number of the select column list items that can be written. SQLN therefore determines the maximum number of elements of array SQLVAR.

■SQLD

SQLD determines the number of valid elements of array SQLVAR.

Executing a output DESCRIBE statement specifying an SQLDA structure when the prepared statement contains select column list items returns the number of select column list items to SQLD.

Executing an input DESCRIBE statement specifying an SQLDA structure when dynamic parameter specifications are written in the prepared statement returns the number of dynamic parameter specifications to SQLD.

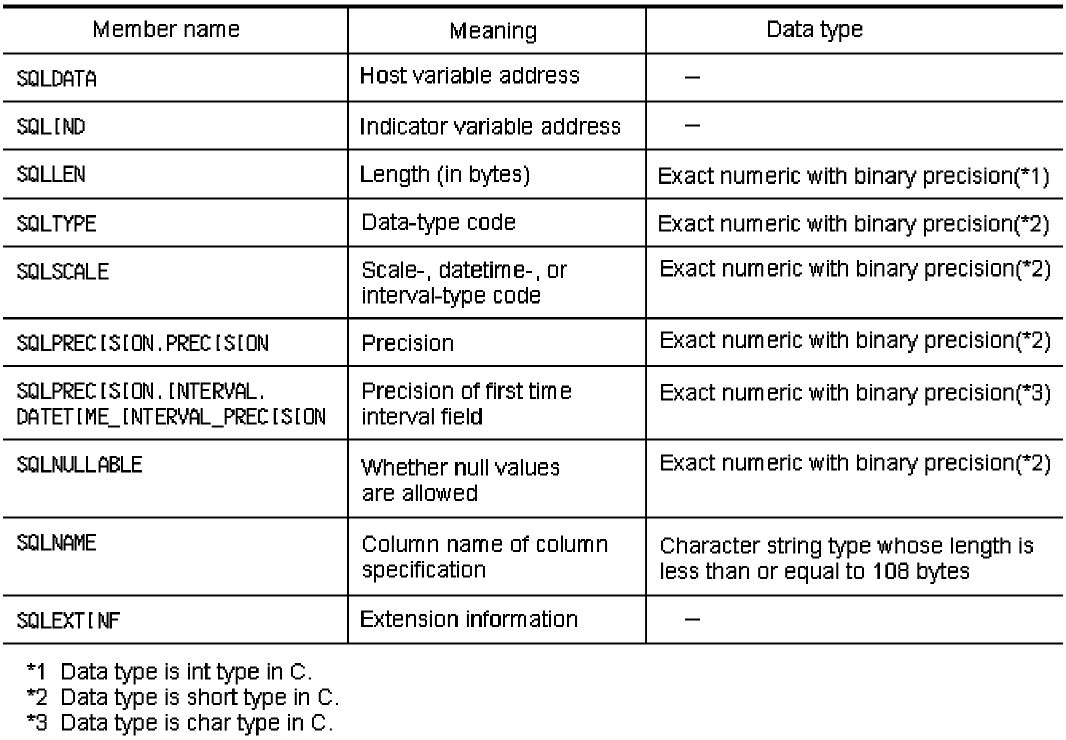
■SQLVAR

SQLVAR is an array that stores various kinds of information for dynamic parameter specifications or select column list items.

Information for this array must be set before an SQL statement is executed for a dynamic parameter specification. An SQL statement in which n dynamic parameter specifications are written is executed as a prepared statement based on the information stored in SQLVA[0] to SQLVAR[n-1].

Information for this array must be set before an SQL statement is executed for a select column list. If the SQL statement in which n select column lists are written is executed as a prepared statement, data is stored in the data buffers specified in SQLVAR[0] to SQLVAR[n-1] after the SQL statement has been executed.

Item descriptor area SQLVAR consists of the items listed in the following table.



Member name	Meaning	Data type
SQLDATA	Host variable address	–
SQLIND	Indicator variable address	–
SQLLEN	Length (in bytes)	Exact numeric with binary precision(*1)
SQLTYPE	Data-type code	Exact numeric with binary precision(*2)
SQLSCALE	Scale-, datetime-, or interval-type code	Exact numeric with binary precision(*2)
SQLPRECISION.PRECISION	Precision	Exact numeric with binary precision(*2)
SQLPRECISION.INTERVAL.DATETIME_INTERVAL_PRECISION	Precision of first time interval field	Exact numeric with binary precision(*3)
SQLNULLABLE	Whether null values are allowed	Exact numeric with binary precision(*2)
SQLNAME	Column name of column specification	Character string type whose length is less than or equal to 108 bytes
SQLEXTINF	Extension information	–

*1 Data type is int type in C.
*2 Data type is short type in C.
*3 Data type is char type in C.

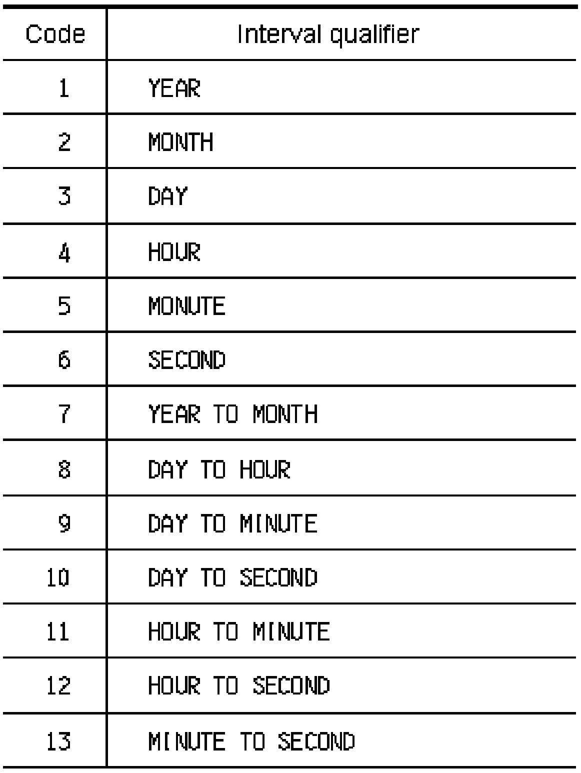
The table below shows the correspondence between the TYPE values (data-type codes) of the SQLDA structure and SQL data types.

Code	Data type
1	CHARACTER
2	NUMERIC
3	DECIMAL
4	INTEGER
5	SMALLINT
7	REAL
8	DOUBLE PRECISION
9	DATE, TIME, or TIMESTAMP
10	INTERVAL
11	NATIONAL CHARACTER
12	CHARACTER VARYING
13	NATIONAL CHARACTER VARYING
30	BLOB
50	ROW_ID

If the TYPE value is 9, indicating the datetime type, any of the following DATE_TIME_INTERVAL_CODE values is set in SQLSCALE:

Code	Data type
1	DATE
2	TIME
3	TIMESTAMP

If the TYPE value is 10, indicating the interval type, any of the following DATETIME_INTERVAL_CODE values is stored in SQLSCALE:

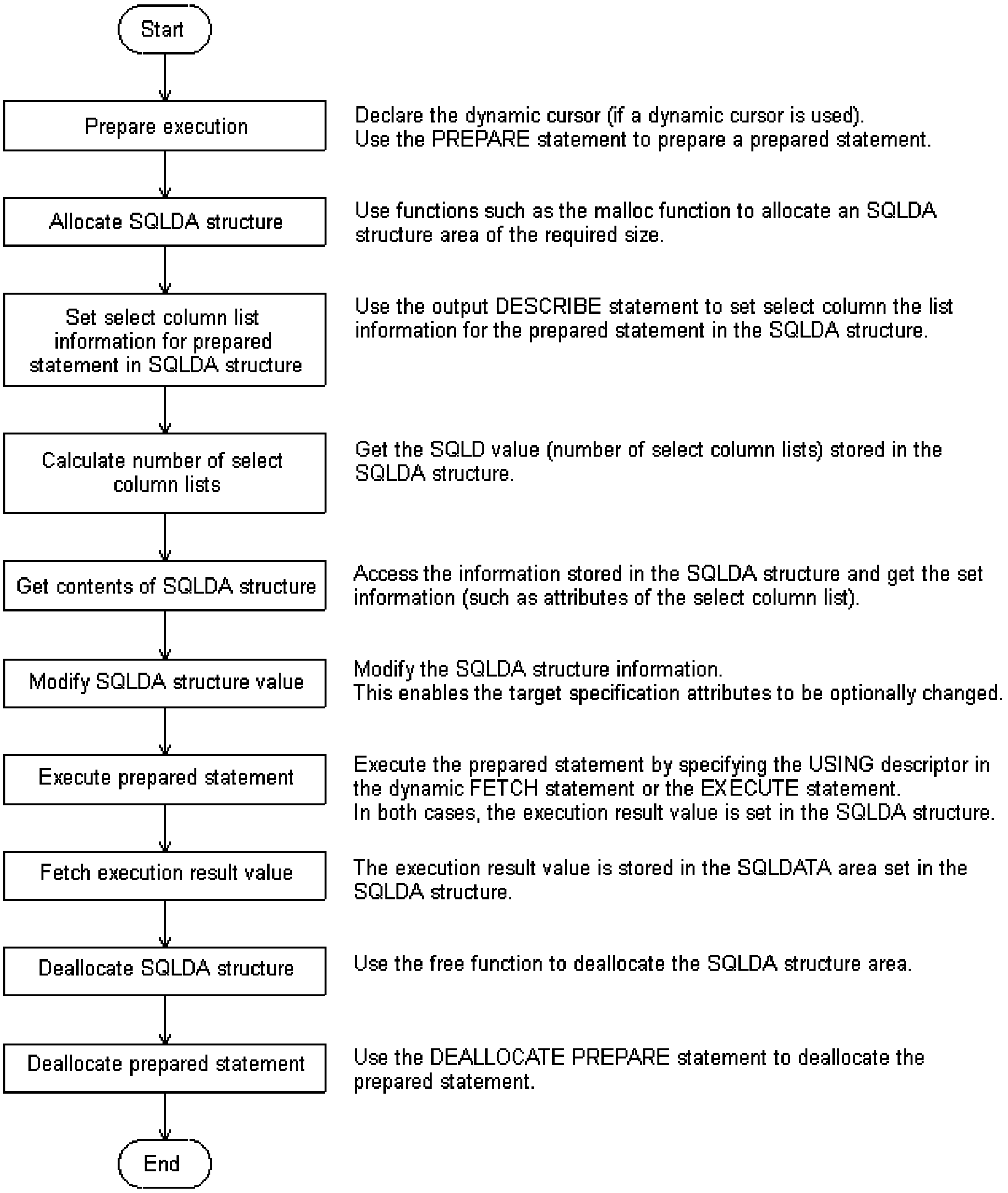


Code	Interval qualifier
1	YEAR
2	MONTH
3	DAY
4	HOUR
5	MONUTE
6	SECOND
7	YEAR TO MONTH
8	DAY TO HOUR
9	DAY TO MINUTE
10	DAY TO SECOND
11	HOUR TO MINUTE
12	HOUR TO SECOND
13	MINUTE TO SECOND

■Procedure for fetching execution results

The order in which execution results are fetched using the USING descriptor is shown in "Figure: Procedure for fetching execution results for a prepared statement." The flow of fetching execution results is explained here. Each SQL statement is explained after "Figure: Procedure for fetching execution results for a prepared statement."

[Figure: Procedure for fetching execution results for a prepared statement]


Start
Prepare execution
Allocate SQLDA structure
Set select column list information for prepared statement in SQLDA structure
Calculate number of select column lists
Get contents of SQLDA structure
Modify SQLDA structure value
Execute prepared statement
Fetch execution result value
Deallocate SQLDA structure
Deallocate prepared statement
End

Declare the dynamic cursor (if a dynamic cursor is used).
Use the PREPARE statement to prepare a prepared statement.

Use functions such as the malloc function to allocate an SQLDA structure area of the required size.

Use the output DESCRIBE statement to set select column the list information for the prepared statement in the SQLDA structure.

Get the SQLD value (number of select column lists) stored in the SQLDA structure.

Access the information stored in the SQLDA structure and get the set information (such as attributes of the select column list).

Modify the SQLDA structure information.
This enables the target specification attributes to be optionally changed.

Execute the prepared statement by specifying the USING descriptor in the dynamic FETCH statement or the EXECUTE statement.
In both cases, the execution result value is set in the SQLDA structure.

The execution result value is stored in the SQLDATA area set in the SQLDA structure.

Use the free function to deallocate the SQLDA structure area.

Use the DEALLOCATE PREPARE statement to deallocate the prepared statement.

When modifying an SQL statement dynamically to manipulate data, prepare execution of the SQL statement.

■Preparing the SQL statement

Use the PREPARE statement to prepare a dynamic SQL statement for execution. To create and execute an SQL statement when the program is executed, store the SQL statement as a character string in a host variable called the SQL statement variable. Next, the PREPARE statement to prepare the SQL statement for execution. At this point, the SQL statement stored in the SQL statement variable is associated with the SQL statement identifier for statement identification.

If the SQL statement variable is a preparable dynamic UPDATE statement (positioned) or a preparable dynamic DELETE statement (positioned), the corresponding cursor must be opened in advance. The specified cursor must also be a dynamic cursor.

If the PREPARE statement is reexecuted for the defined SQL statement identifier, the allocated prepared statement is deallocated and a new prepared statement is prepared. If the allocated prepared statement is a dynamic SELECT statement, the statement prepared for a preparable dynamic UPDATE or DELETE statement (positioned) that references the cursor corresponding to the SQL statement identifier is also deallocated. However, if the prepared statement is a dynamic SELECT statement, the cursor corresponding to the SQL statement identifier must have already been closed.

An example of specifying the PREPARE statement is shown below. The SQL statement to be prepared for execution is assumed to be stored in SQL statement variable str. The SQL statement identifier of this SQL statement is defined as STM.

$$\texttt{PREPARE}\quad \underbrace{\texttt{STM}}_{\substack{\text{SQL statement}\\ \text{identifier}}}\quad \texttt{FROM}\quad \underbrace{\texttt{:str}}_{\substack{\text{SQL statement}\\ \text{variable}}}$$

■Deallocating an SQL statement

Use the DEALLOCATE PREPARE statement to deallocate the prepared statement prepared by the PREPARE statement.

If the prepared statement is a dynamic SELECT statement, the statement prepared for a preparable dynamic UPDATE or DELETE statement (positioned) that references the cursor corresponding to the SQL statement identifier is also deallocated. In this case, the cursor must have already been closed.

An example of specifying the DEALLOCATE PREPARE statement is shown below. The prepared statement for which the PREPARE statement defined the SQL statement identifier as STM is assumed to be the prepared statement that will be deallocated.

$$\texttt{DEALLOCATE}\quad \texttt{PREPARE}\quad \underbrace{\texttt{STM}}_{\text{SQL statement identifier}}$$

■SQL statements used to manipulate the SQLDA structure

As shown in "Figure: Procedure for fetching execution results for a prepared statement," use the DESCRIBE statement to fetch select column list information from the database to the SQLDA structure. The DESCRIBE statement and its functions are explained below.

◆DESCRIBE statement

The DESCRIBE statement fetches select column list information for the prepared statement prepared by the PREPARE statement to the SQLDA structure. This type of statement is called an output DESCRIBE statement. The following is an example of the DESCRIBE statement that fetches select column list information for the prepared statement corresponding to SQL statement identifier STM2 to the SQLDA structure.

$$\texttt{DESCRIBE}\quad \texttt{OUTPUT}\quad \underbrace{\texttt{STM2}}_{\text{SQL statement identifier}}\quad \texttt{USING}\quad \texttt{SQL}\quad \texttt{DESCRIPTOR}\quad \underbrace{\texttt{:OUTSQLDA}}_{\text{SQLDA structure}}$$

If the prepared statement for SQL statement identifier STM2 is the statement shown below, the contents of the SQLDA structure are as follows:

SELECT PRODUCT. STOCKQTY FROM STOCKMN. STOCK WHERE ITEMNO = 110

Select column list

- SQLD: 2
  The number of select column lists is set to 2.

- Member information in the item descriptor area of the SQLDA structure
  The attributes of the product name (PRODNAME) and stock quantity (STOCKQTY) are set as follows:
  - Product name: NATIONAL CHARACTER(10) (with NOT NULL constraint)
  - Stock quantity: INTEGER (without NOT NULL constraint)

Identifier	PRODUCT	STOCKQTY
SQLTYPE	1	4
SQLLEN	20	4
SQLPRECISION.PRECISION	0	31
SQLSCALE	0	0
SQLNULLABLE	0	1
SQLIND	No value is set.	No value is set.
SQLDATA	No value is set.	No value is set.
SQLNAME	Product name(*1)	Stock quantity(*1)
SQLPRECISION.INTERVAL	0	0

*1 SQLNAME is set only by the SQLLEN length; none is set in the remaining area.

7.2.1.1 Preparing and executing dynamic SELECT statements (for SQLDA structure)

To fetch data continuously, prepare and execute a dynamic SELECT statement.

An example of an application program that executes a dynamic SELECT statement using the SQLDA structure is shown in "Figure: Example of application program that executes a dynamic SELECT statement."

Example:

This is an example of a dynamic SELECT statement entered from a terminal.

[Figure: Example of application program that executes a dynamic SELECT statement]

(Continued)

```
#include <stdio.h>
#include <string.h>
#include "qdbnosis.h"

EXEC SQL BEGIN DECLARE SECTION;
    char      SQLSTATE[6];
    char      SQLMSG[256];
    VARCHAR   str[1024];                     /* SQL statement variable        */   ①
    SQLDA     *sqlda_out;                    /* SQLDA structure               */
EXEC SQL END DECLARE SECTION;


void main()
 {
    short     vartype;                       /* Data type                     */
    short     varleng;                       /* Number of characters          */
    short     varolen;                       /* Number of bytes               */
    short     varprec;                       /* Precision                     */
    short     varscal;                       /* Scale      SQL descriptor area */
    char      varname[129];                  /* Column name                   */
    short     vardcod;                       /* Date and time interval        */
    short     vardpre;                       /* Interval precision            */
    short     varocount;                     /* Number of select column lists */
    short     i;                             /* Counter for SQL descriptor area */
    char      ocdata[100][15];               /* CHARACTER type                */
    char      oncdata[100][21];              /* NCHAR type                    */
    long      oidata[100];                   /* Exact numeric type   Target specification*/
    double    ofdata[100];                   /* Approximate numeric type      */
    char      oddata[100][22];               /* Date and time interval        */
    short     inddata[100];                  /* Indicator variable            */
    size_t    size;                          /* SQLDA size                    */


    memset(SQLSTATE, 0x00, 6);
    memset(SQLMSG, 0x00, 256);
    memset(&str, 0x00, sizeof(str));
    memset(varname, 0x00, sizeof(varname));                                          ②
    memset(ocdata, 0x00, sizeof(ocdata) );
    memset(oncdata, 0x00, sizeof(oncdata) );
    memset(oddata, 0x00, sizeof(oddata));
```

(Continued)

```
    size = sizeof(SQLDA) + sizeof(sqlvar) * (100 - 1);                      ③
    sqlda_out = (SQLDA *)malloc(size);
    memset( sqlda_out, 0x00, size );
    sqlda_out->SQLN = 100;                                                   ④


    EXEC SQL CONNECT TO DEFAULT;                                             ⑤
    EXEC SQL WHENEVER SQLERROR GOTO :ERR;                                    ⑥
    printf( "Specify the dynamic SELECT statement to be executed. \n" );
    gets( str.sqlvar );                                                      ⑦
    str.sqllen = strlen( str.sqlvar );
    EXEC SQL DECLARE CUR1 CURSOR FOR STMID;                                  ⑧
    EXEC SQL PREPARE STMID FROM :str;                                        ⑨
    EXEC SQL DESCRIBE OUTPUT STMID USING SQL DESCRIPTOR :sqlda_out;          ⑩
    /**************/
    /* Select column list */
    /**************/
    varocount = sqlda_out->SQLD;                                             ⑪

 for( i = 0; i < varocount; i++ ) {
        vartype = sqlda_out->SQLVAR[i].SQLTYPE;
        varleng = sqlda_out->SQLVAR[i].SQLLEN;
        varprec = sqlda_out->SQLVAR[i].SQLPRECISION.PRECISION;                 ⑫
        varscal = sqlda_out->SQLVAR[i].SQLSCALE;
        vardcod = sqlda_out->SQLVAR[i].SQLSCALE;
        vardpre = sqlda_out->SQLVAR[i].SQLPRECISION.INTERVAL.
        DATETIME_INTERVAL_PRECISION;
        /* CHAR type or VCHAR type */
        if( vartype == 1 || vartype == 12 ) {
            sqlda_out->SQLVAR[i].SQLTYPE = 1;
            sqlda_out->SQLVAR[i].SQLLEN = 14;
            sqlda_out->SQLVAR[i].SQLDATA = (char *)&ocdata[i][0];
        }
        /* NCHAR type or NVCHAR type */
        else if( vartype == 11 || vartype == 13 ) {
            sqlda_out->SQLVAR[i].SQLTYPE = 11;
            sqlda_out->SQLVAR[i].SQLLEN = 20;
            sqlda_out->SQLVAR[i].SQLDATA = (char *)&oncdata[i][0];
        }                                                                      ⑬
        /* INTEGER type or SMALLINT type   */
        else if( vartype == 4 || vartype == 5 ) {
            sqlda_out->SQLVAR[i].SQLTYPE = 4;
            sqlda_out->SQLVAR[i].SQLLEN  = 4;
            sqlda_out->SQLVAR[i].SQLPRECISION.PRECISION = 31;
            sqlda_out->SQLVAR[i].SQLSCALE = 0;
            sqlda_out->SQLVAR[i].SQLDATA = (char *)&oidata[i];
        }
```

(Continued)

```
        /*  NUMERIC type, DECIMAL type, REAL type, or DOUBLE PRECISION type   */
        else if( vartype == 2 || vartype == 3 ||
                 vartype == 7 || vartype == 8 ) {
            sqlda_out->SQLVAR[i].SQLTYPE = 8;
            sqlda_out->SQLVAR[i].SQLLEN  = 8;
            sqlda_out->SQLVAR[i].SQLPRECISION.PRECISION = 52;
            sqlda_out->SQLVAR[i].SQLSCALE = 0;
            sqlda_out->SQLVAR[i].SQLDATA = (char *)&ofdata[i];
        }
        /*  Datetime type or INTERVAL type   */
        else if( vartype == 9 || vartype == 10 ) {
            sqlda_out->SQLVAR[i].SQLTYPE = 12;
            sqlda_out->SQLVAR[i].SQLLEN = 21;
            sqlda_out->SQLVAR[i].SQLDATA = (char *)&oddata[i][0];
        }
    }
    EXEC SQL OPEN CUR1;                                                  ⑭
    EXEC SQL WHENEVER NOT FOUND GOTO :NOTFND;                            ⑮
    for(;;) {
        EXEC SQL FETCH CUR1 INTO SQL DESCRIPTOR :sqlda_out;              ⑯
        /*************/
        /*Execution result */
        /*************/
        for( i = 0; i < varocount; i++ ) {
            vartype = sqlda_out->SQLVAR[i].SQLTYPE;

            memcpy( varname, sqlda_out->SQLVAR[i].SQLNAME.SQLNAMEC,
                             sqlda_out->SQLVAR[i].SQLNAME.SQLNAMEL );
            varname[sqlda_out->SQLVAR[i].SQLNAME.SQLNAMEL] = 0x00;
            /* CHAR type    */
            if( vartype == 1 ) {
                printf( "%s -> %s   ", varname, &ocdata[i][0] );
            }
            /* NCHAR type   */
            else if( vartype == 11 ) {
                printf( "%s -> %s   ", varname, &oncdata[i][0] );
            }
            /* Exact numeric type   */
            else if( vartype == 4 ) {
                printf( "%s -> %d   ", varname, oidata[i] );
            }
            /* Approximate numeric type     */
            else if( vartype == 8 ) {
                printf( "%s -> %lf  ", varname, ofdata[i] );
            }
```

(Continued)

```
            /* Datetime type or INTERVAL type    */
            else if( vartype == 12 ) {
                printf( "%s -> %s   ", varname, &oddata[i][2] );
            }
        }
        printf( "\n" );
    }
NOTFND:
    EXEC SQL CLOSE CUR1;                                                  ⑰
    EXEC SQL DEALLOCATE PREPARE STMID;                                    ⑱
    EXEC SQL COMMIT WORK;                                                 ⑲
    EXEC SQL DISCONNECT DEFAULT;                                          ⑳
    free( sqlda_out );
    return;
ERR:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    printf( "SQLERROR SQLSTATE = %s SQLMSG = %s\n", SQLSTATE, SQLMSG );
    EXEC SQL ROLLBACK WORK;
    EXEC SQL DISCONNECT DEFAULT;
    free( sqlda_out );
    return;
}
```

① The SQL statement variable is the area in which the SQL statement is stored. A variable-length character string is expanded into the following structure when the program is compiled:

```
struct {
    short  sqllen;          /* Length area variable           */
    char   sqlvar[1024];    /* Character string area variable */
}str;
```

② Initializes variables.
③ Allocates an SQLDA structure area.
④ Specifies the number of SQLVAR arrays (SQLN value).
⑤ Opens a connection.
⑥ If an error other than "no data" occurs during execution of the SQL statement, the program branches to ERR, outputs error information, and terminates.
⑦ Reads the dynamic SELECT statement entered from the terminal and stores it in str.
⑧ Declares cursor CUR1. Specifies SQL statement identifier STMID as the cursor specification.
⑨ Prepares the SQL statement in str for execution.
⑩ Fetches the select column list information for the entered dynamic SELECT statement to the SQLDA structure.
⑪ Gets the number of the select column lists (SQLD value) set in the SQLDA structure.
⑫ Gets the information (values of SQLTYPE, SQLLEN, SQLPRECISION, and SQLSCALE) for each select column list set in the SQLDA structure.
⑬ Sets the data types and addresses of variables of the target specification that fetches execution results to the SQLDA structure.
  - Changes CHARACTER(n) and CHARACTER VARYING(n) to CHARACTER(14). (n indicates the length of the character string.)
  - Changes NATIONAL CHARACTER(n) and NATIONAL CHARACTER VARYING(n) to NATIONAL CHARACTER(10). (n indicates the length of the character string.)
  - Changes INTEGER and SMALLINT to INTEGER.
  - Changes NUMERIC(p, q), DECIMAL(p, q), REAL, and DOUBLE PRECISION to DOUBLE PRECISION. (p indicates the precision and q indicates the scale.)
  - Changes DATE, TIME, TIMESTAMP, and INTERVAL to CHARACTER VARYING(19). (n indicates the length of the character string.)

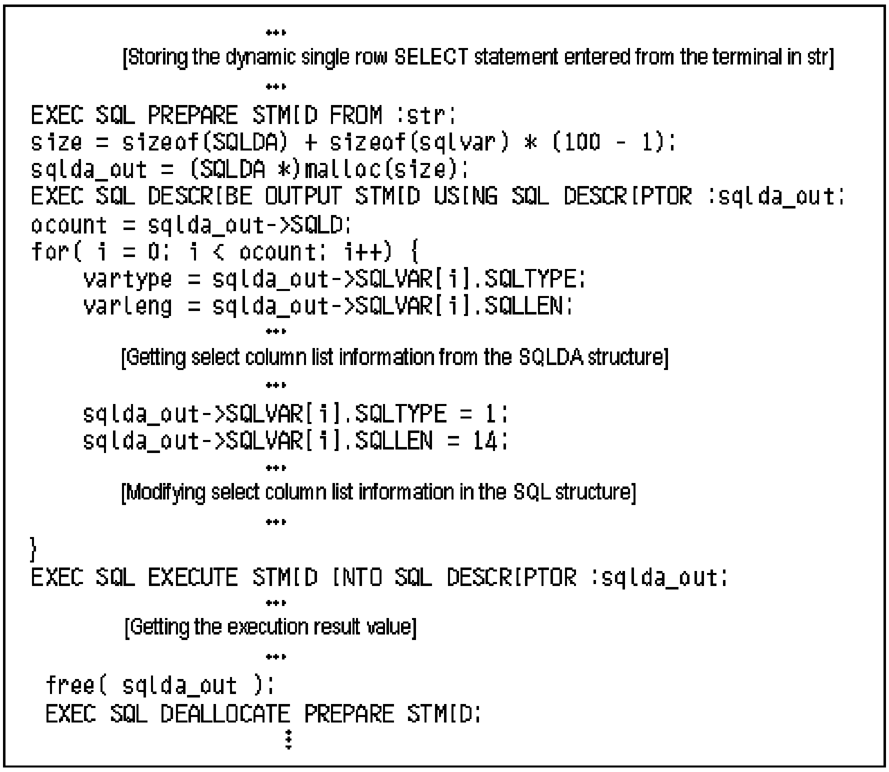
⑭ Opens cursor CUR1.
⑮ If search result data is not found, the program branches to NOTFND and terminates.
⑯ Positions cursor CUR1 and reads data. The execution result value is set at the address set in the SQLDATA field of the SQLDATA structure.
⑰ Closes cursor CUR1.
⑱ Deallocates the prepared statement corresponding to SQL statement identifier STMID.
⑲ Terminates the current transaction.
⑳ Closes the connection.

7.2.1.2 Preparing and executing dynamic single row SELECT statements (for SQLDA structure)

To fetch data from one row, prepare and execute a dynamic single row SELECT statement. To execute a dynamic single row SELECT statement, use an EXECUTE statement in which the USING clause is specified. An example of a dynamic single row SELECT statement using the SQLDA structure is shown below. For information about the SQL statements that can use the SQL structure, see the examples in "Figure: Procedure for fetching execution results for a prepared statement."

Example:

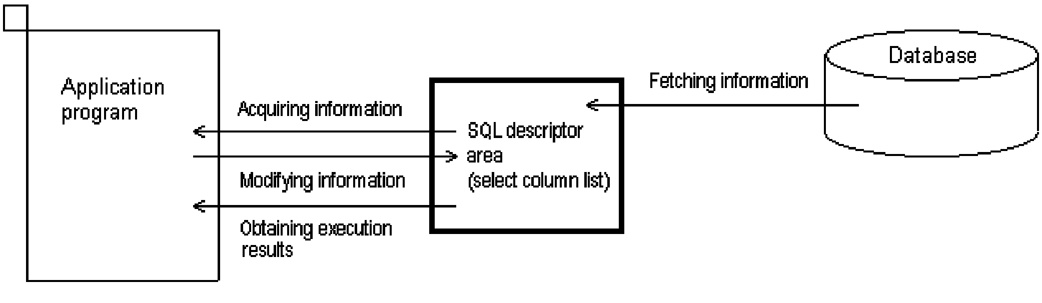
This is an example of a dynamic single row SELECT statement entered from a terminal.



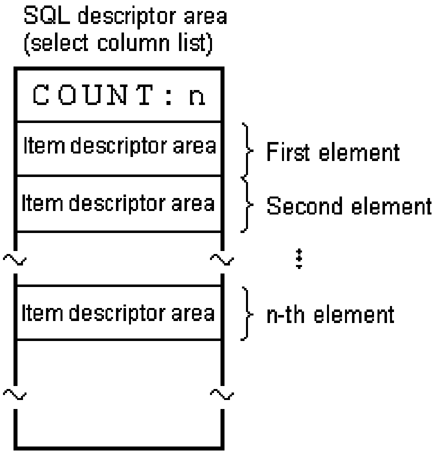
```
                  ...
       [Storing the dynamic single row SELECT statement entered from the terminal in str]
                  ...
EXEC SQL PREPARE STMID FROM :str;
size = sizeof(SQLDA) + sizeof(sqlvar) * (100 - 1);
sqlda_out = (SQLDA *)malloc(size);
EXEC SQL DESCRIBE OUTPUT STMID USING SQL DESCRIPTOR :sqlda_out;
ocount = sqlda_out->SQLD;
for( i = 0; i < ocount; i++) {
    vartype = sqlda_out->SQLVAR[i].SQLTYPE;
    varleng = sqlda_out->SQLVAR[i].SQLLEN;
                  ...
       [Getting select column list information from the SQLDA structure]
                  ...
    sqlda_out->SQLVAR[i].SQLTYPE = 1;
    sqlda_out->SQLVAR[i].SQLLEN = 14;
                  ...
       [Modifying select column list information in the SQL structure]
                  ...
}
EXEC SQL EXECUTE STMID INTO SQL DESCRIPTOR :sqlda_out;
                  ...
       [Getting the execution result value]
                  ...
 free( sqlda_out );
 EXEC SQL DEALLOCATE PREPARE STMID;
                   ⋮
```

7.2.2 Preparing SQL statements and manipulating the SQL descriptor area

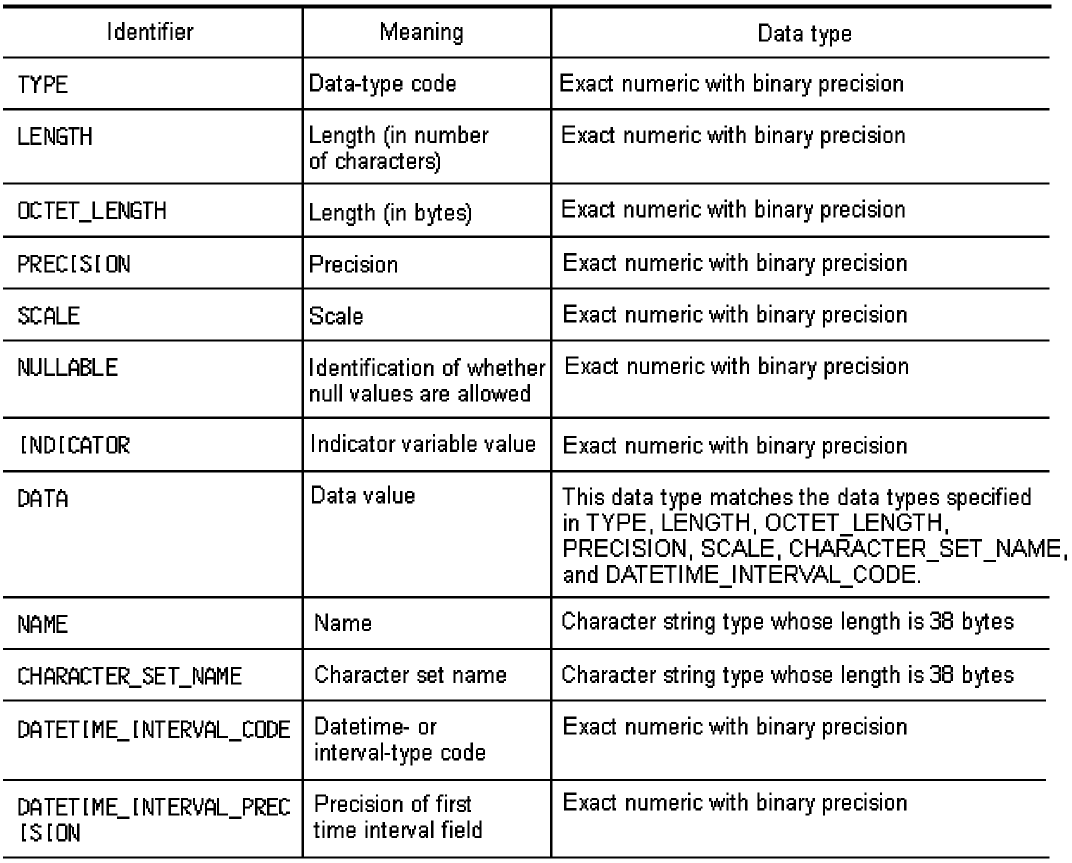
When data is fetched from a database using dynamic SQL, specify the target that will receive the execution results. For the specified target to be able to receive the execution results, the select column list information must match the attributes of the specified target. Matching is accomplished by fetching the select column list information from the database to the SQLDA structure, then modifying the attributes. The SQL descriptor area is positioned as shown in the figure below.


Application program
Acquiring information
Modifying information
Obtaining execution results
SQL descriptor area (select column list)
Fetching information
Database

The SQL descriptor area either consists of only one identifier (COUNT) or consists of one identifier (COUNT) and one or more item descriptor areas (elements of the SQL descriptor area). COUNT indicates the number of select column lists in the SQL descriptor area; the data type is an exact numeric with binary precision. In the following figure, n indicates the number of select column lists in the SQL descriptor area.

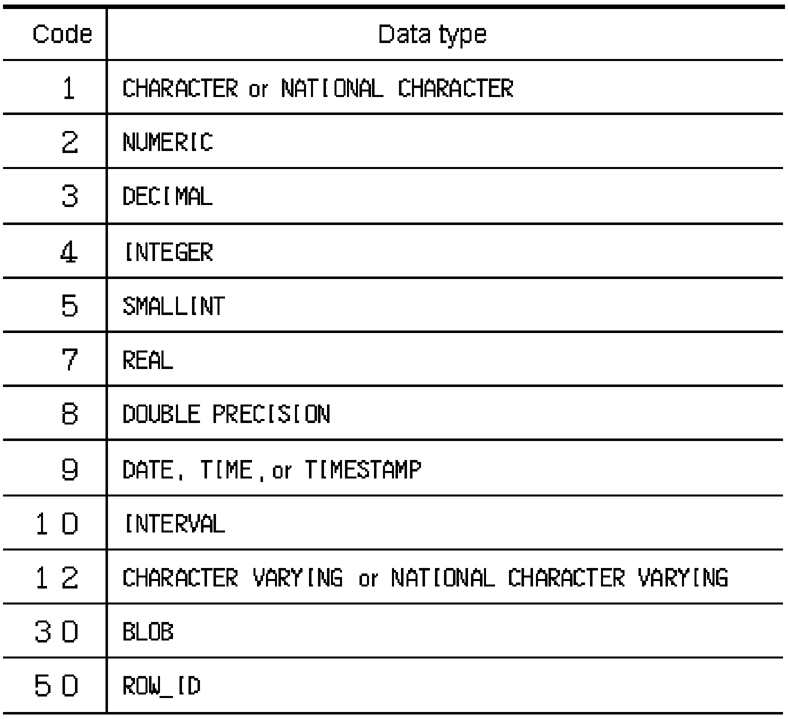


Each item descriptor area consists of the items listed in the table below.



Identifier	Meaning	Data type
TYPE	Data-type code	Exact numeric with binary precision
LENGTH	Length (in number of characters)	Exact numeric with binary precision
OCTET_LENGTH	Length (in bytes)	Exact numeric with binary precision
PRECISION	Precision	Exact numeric with binary precision
SCALE	Scale	Exact numeric with binary precision
NULLABLE	Identification of whether null values are allowed	Exact numeric with binary precision
INDICATOR	Indicator variable value	Exact numeric with binary precision
DATA	Data value	This data type matches the data types specified in TYPE, LENGTH, OCTET_LENGTH, PRECISION, SCALE, CHARACTER_SET_NAME, and DATETIME_INTERVAL_CODE.
NAME	Name	Character string type whose length is 38 bytes
CHARACTER_SET_NAME	Character set name	Character string type whose length is 38 bytes
DATETIME_INTERVAL_CODE	Datetime- or interval-type code	Exact numeric with binary precision
DATETIME_INTERVAL_PRECISION	Precision of first time interval field	Exact numeric with binary precision

The table below shows the correspondence between the TYPE values (data-type codes) of the SQL descriptor area and SQL data types.

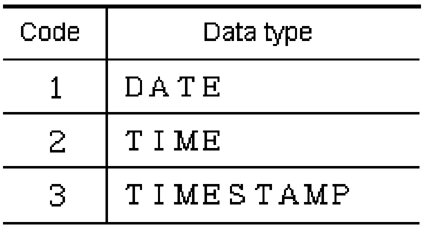


Code	Data type
１	CHARACTER or NATIONAL CHARACTER
２	NUMERIC
３	DECIMAL
４	INTEGER
５	SMALLINT
７	REAL
８	DOUBLE PRECISION
９	DATE, TIME, or TIMESTAMP
１０	INTERVAL
１２	CHARACTER VARYING or NATIONAL CHARACTER VARYING
３０	BLOB
５０	ROW_ID

If the TYPE value is 1 or 12, the CHARACTER_SET_NAME value (character set name) varies according to character string type and national character string type.

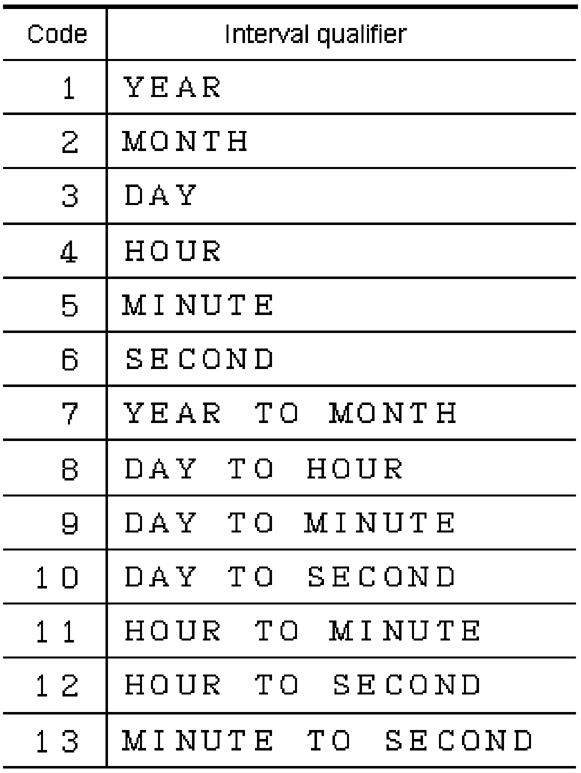
- Character string type: BASIC
- National character string type: NCHAR

If the TYPE value is 9, indicating the datetime type, the DATETIME_INTERVAL_CODE value is any of the codes listed in the table below.



Code	Data type
１	ＤＡＴＥ
２	ＴＩＭＥ
３	ＴＩＭＥＳＴＡＭＰ

If the TYPE value is 10, indicating the interval type, the DATETIME_INTERVAL_CODE value is any of the following codes listed in the table below.

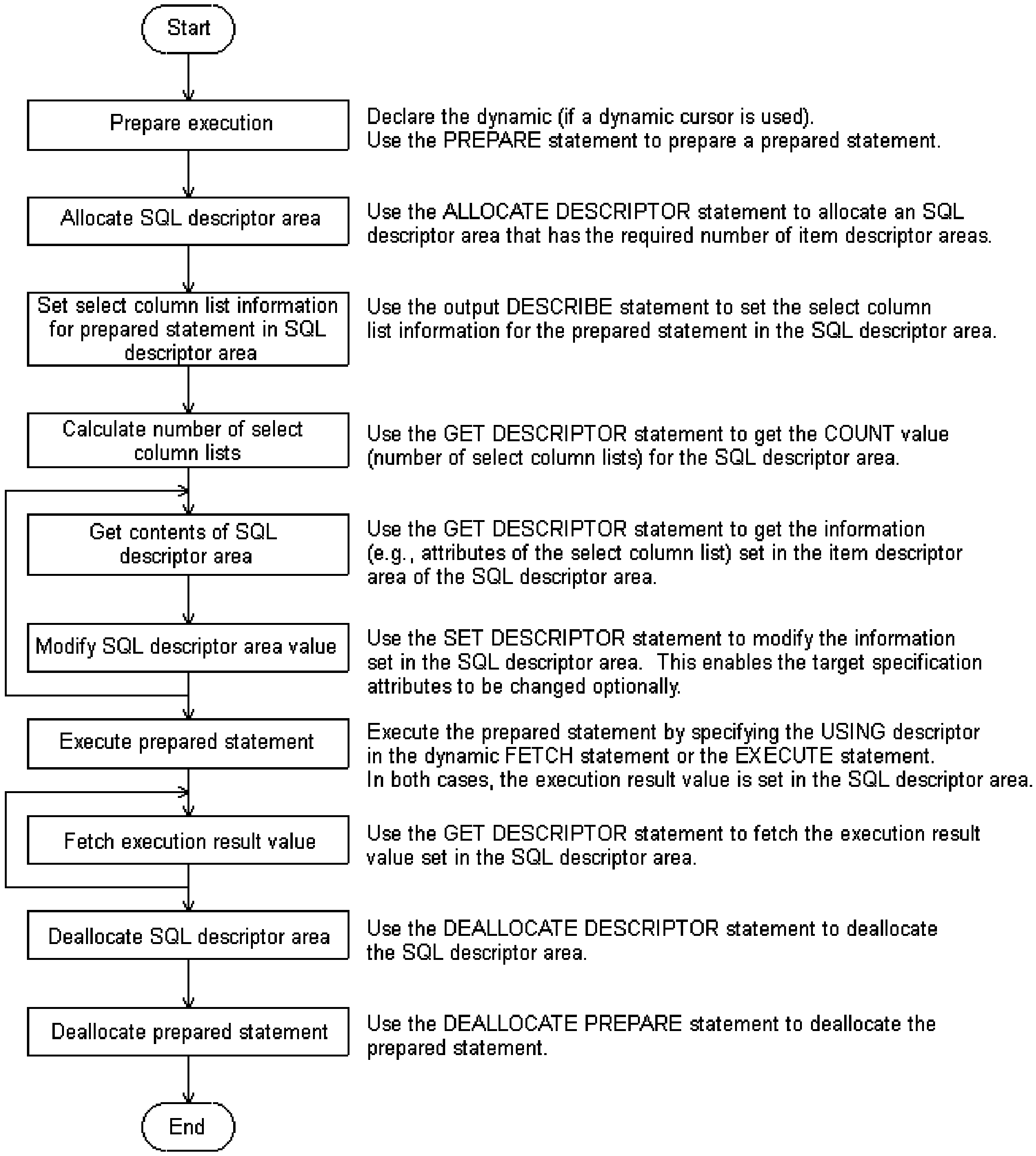


Code	Interval qualifier
1	YEAR
2	MONTH
3	DAY
4	HOUR
5	MINUTE
6	SECOND
7	YEAR TO MONTH
8	DAY TO HOUR
9	DAY TO MINUTE
10	DAY TO SECOND
11	HOUR TO MINUTE
12	HOUR TO SECOND
13	MINUTE TO SECOND

■Procedure for fetching execution results

The order in which execution results are fetched using the USING descriptor is shown in "Figure: Procedure for fetching execution results for a prepared statement." The flow of fetching execution results is explained here. Each SQL statement is explained after "Figure: Procedure for fetching execution results for a prepared statement."

[Figure: Procedure for fetching execution results for a prepared statement]


Start
Prepare execution
Allocate SQL descriptor area
Set select column list information for prepared statement in SQL descriptor area
Calculate number of select column lists
Get contents of SQL descriptor area
Modify SQL descriptor area value
Execute prepared statement
Fetch execution result value
Deallocate SQL descriptor area
Deallocate prepared statement
End

Declare the dynamic (if a dynamic cursor is used).
Use the PREPARE statement to prepare a prepared statement.

Use the ALLOCATE DESCRIPTOR statement to allocate an SQL descriptor area that has the required number of item descriptor areas.

Use the output DESCRIBE statement to set the select column list information for the prepared statement in the SQL descriptor area.

Use the GET DESCRIPTOR statement to get the COUNT value (number of select column lists) for the SQL descriptor area.

Use the GET DESCRIPTOR statement to get the information (e.g., attributes of the select column list) set in the item descriptor area of the SQL descriptor area.

Use the SET DESCRIPTOR statement to modify the information set in the SQL descriptor area. This enables the target specification attributes to be changed optionally.

Execute the prepared statement by specifying the USING descriptor in the dynamic FETCH statement or the EXECUTE statement.
In both cases, the execution result value is set in the SQL descriptor area.

Use the GET DESCRIPTOR statement to fetch the execution result value set in the SQL descriptor area.

Use the DEALLOCATE DESCRIPTOR statement to deallocate the SQL descriptor area.

Use the DEALLOCATE PREPARE statement to deallocate the prepared statement.

When modifying an SQL statement dynamically to manipulate data, prepare execution of the SQL statement.

■Preparing the SQL statement

Use the PREPARE statement to prepare a dynamic SQL statement for execution. To create and execute an SQL statement when the program is executed, store the SQL statement as a character string in a host variable called the SQL statement variable. Next, the PREPARE statement to prepare the SQL statement for execution. At this point, the SQL statement stored in the SQL statement variable is associated with the SQL statement identifier for statement identification.

If the SQL statement variable is a preparable dynamic UPDATE statement (positioned) or a preparable dynamic DELETE statement (positioned), the corresponding cursor must be opened in advance. The specified cursor must also be a dynamic cursor.

If the PREPARE statement is reexecuted for the defined SQL statement identifier, the allocated prepared statement is deallocated and a new prepared statement is prepared. If the allocated prepared statement is a dynamic SELECT statement, the statement prepared for a preparable dynamic UPDATE or DELETE statement (positioned) that references the cursor corresponding to the SQL statement identifier is also deallocated. However, if the prepared statement is a dynamic SELECT statement, the cursor corresponding to the SQL statement identifier must have already been closed.

An example of specifying the PREPARE statement is shown below. The SQL statement to be prepared for execution is assumed to be stored in SQL statement variable str. The SQL statement identifier of this SQL statement is defined as STM.

$$\texttt{PREPARE}\ \ \underbrace{\texttt{STM}}_{\substack{\text{SQL statement}\\ \text{identifier}}}\ \ \texttt{FROM}\ \ \underbrace{\texttt{:str}}_{\substack{\text{SQL statement}\\ \text{variable}}}$$

■Deallocating an SQL statement

Use the DEALLOCATE PREPARE statement to deallocate the prepared statement prepared by the PREPARE statement.

If the prepared statement is a dynamic SELECT statement, the statement prepared for a preparable dynamic UPDATE or DELETE statement (positioned) that references the cursor corresponding to the SQL statement identifier is also deallocated. In this case, the cursor must have already been closed.

An example of specifying the DEALLOCATE PREPARE statement is shown below. The prepared statement for which the PREPARE statement defined the SQL statement identifier as STM is assumed to be the prepared statement that will be deallocated.

$$\texttt{DEALLOCATE}\ \ \texttt{PREPARE}\ \ \underbrace{\texttt{STM}}_{\text{SQL statement identifier}}$$

■SQL statements used to manipulate the SQL descriptor area

As shown in "Figure: Procedure for fetching execution results for a prepared statement," use SQL statements to fetch select column list information from the database to the SQL descriptor area and to fetch the select column list information placed in the SQL descriptor area. The SQL statements used to manipulate the SQL descriptor area and their functions are explained below.

◆ALLOCATE DESCRIPTOR statement

The ALLOCATE DESCRIPTOR statement allocates the SQL descriptor area for the specified descriptor name. The allocated SQL descriptor area contains the number of item descriptor areas specified in this instance.

An example of allocating the SQL descriptor area with descriptor name DESC1 is shown below. In this example, the number of item descriptor areas in the SQL descriptor area is assumed to be 200.

$$\texttt{ALLOCATE}\ \ \texttt{DESCRIPTOR}\ \ \underbrace{\texttt{'DESC1'}}_{\text{Descriptor name}}\ \ \texttt{WITH MAX}\ \ \underbrace{\texttt{200}}_{\text{This instance}}$$

◆DEALLOCATE DESCRIPTOR statement

The DEALLOCATE DESCRIPTOR statement deallocates the SQL descriptor area allocated by the ALLOCATE DESCRIPTOR statement. If the descriptor name specified by the DEALLOCATE DESCRIPTOR statement is also specified in a dynamic OPEN statement, the cursor must have already been closed.

An example of deallocating the SQL descriptor area with descriptor name DESC1 is shown below.

```
DEALLOCATE DESCRIPTOR 'DESC1'
```

Descriptor name

◆DESCRIBE statement

The DESCRIBE statement fetches select column list information for the prepared statement prepared by the PREPARE statement to the SQL descriptor area. This type of statement is called an output DESCRIBE statement. The following is an example of the DESCRIBE statement that fetches select column list information for the prepared statement corresponding to SQL statement identifier STM2 to the SQL descriptor area.

```
DESCRIBE OUTPUT STM2 USING SQL DESCRIPTOR 'DESC2'
```

SQL statement identifier　　Descriptor name

If the prepared statement for SQL statement identifier STM2 is the statement shown below, the contents of SQL descriptor area DESC2 are as follows:

```
SELECT PRODUCT. STOCKQTY FROM STOCKMN. STOCK WHERE ITEMNO = 110
```

Select column list

● COUNT: 2
The number of select column lists is set to 2.

● Item descriptor area
The attributes of the product name (PRODNAME) and stock quantity (STOCKQTY) are set as follows:

- Product name: NATIONAL CHARACTER(10) (with NOT NULL constraint)
- Stock quantity: INTEGER (without NOT NULL constraint)

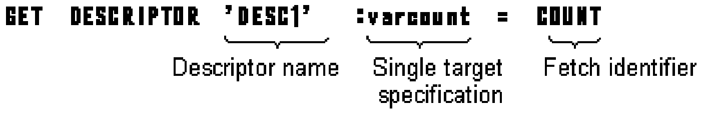
Identifier	PRODUCT	STOCKQTY
TYPE	1	4
LENGTH	10	4
OCTET_LENGTH	20	4
PRECISION	0	31
SCALE	0	0
NULLABLE	0	1
INDICATOR	No value is set.	No value is set.
DATA	No value is set.	No value is set.
NAME	Product name(*1)	Stock quantity(*1)
CHARACTER_SET_NAME	NCHAR(*1)	35-byte space
DATETIME_INTERVAL_CODE	0	0
DATETIME_INTERVAL_PRECISION	0	0

*1 If this item is less than 38 bytes, the remaining area is padded with blanks.

◆GET DESCRIPTOR statement

The GET DESCRIPTOR statement fetches the information set in the SQL descriptor area to a host variable. The data type of the host variable must match data type of the corresponding fetch identifier.

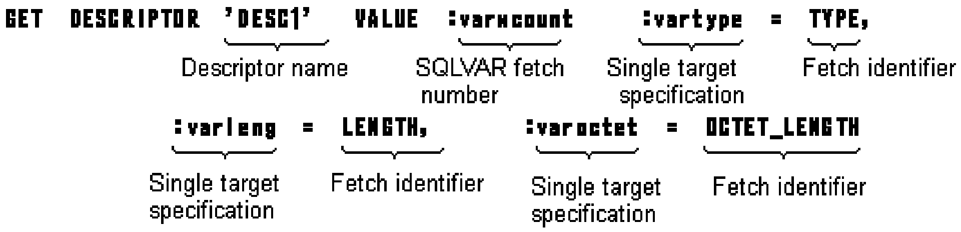
An example of fetching the number of select column lists to host variable varcount is shown below.



```
GET DESCRIPTOR 'DESC1' :varcount = COUNT
```

Descriptor name
Single target specification
Fetch identifier

The following is an example of fetching the contents of the select column lists set in the SQL descriptor area to host variables vartypc, varleng, and varoctct. In this example, the values set in varwcount are assumed to be the element numbers of the item descriptor areas whose information is being fetched.

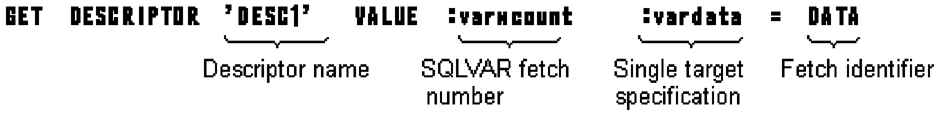


```
GET DESCRIPTOR 'DESC1' VALUE :varwcount :vartype = TYPE,
               :varleng = LENGTH,  :varoctet = OCTET_LENGTH
```

Descriptor name
SQLVAR fetch number
Single target specification
Fetch identifier
Single target specification
Fetch identifier
Single target specification
Fetch identifier

When execution results are fetched from the SQL descriptor area, the data types of the host variables must correspond to TYPE, LENGTH, OCTET_LENGTH, PRECISION, SCALE, and CHARACTER_SET_NAME of the corresponding item descriptor area.

An example of fetching the execution results set in the SQL descriptor area to host variable vardata is shown below. In this example, the value set in varwcount is assumed to be the element number of the item descriptor area whose information is to be fetched.

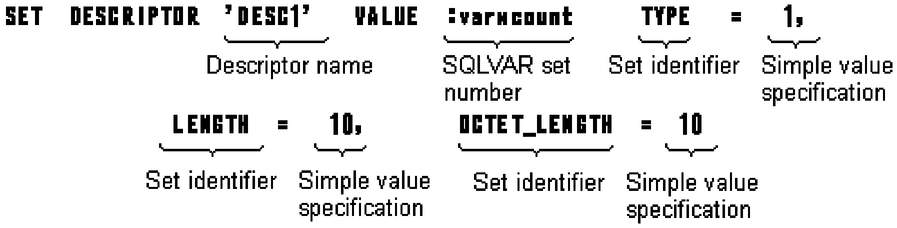


```
GET DESCRIPTOR 'DESC1' VALUE :varwcount :vardata = DATA
```

Descriptor name
SQLVAR fetch number
Single target specification
Fetch identifier

◆SET DESCRIPTOR statement

The SET DESCRIPTOR statement sets the data type of the target specification used to fetch execution results in the select column list of the SQL descriptor area. The data types of the host variables must correspond to the set identifiers. An example of changing the data type of the select column list of the SQL descriptor area is shown below. In this example, since CHARACTER(11) is assumed to be declared as the data type of host variable vardata, which receives the execution results, change the data type of the select column list to CHARACTER(10). Also in this example, the value set in varwcount is assumed to be the element number of the item descriptor area that will be set.



```
SET DESCRIPTOR 'DESC1' VALUE :varxcount TYPE = 1,
LENGTH = 10, OCTET_LENGTH = 10
```

Descriptor name
SQLVAR set number
Set identifier
Simple value specification
Set identifier
Simple value specification
Set identifier
Simple value specification

7.2.2.1 Preparing and executing dynamic SELECT statements (for SQL descriptor area)

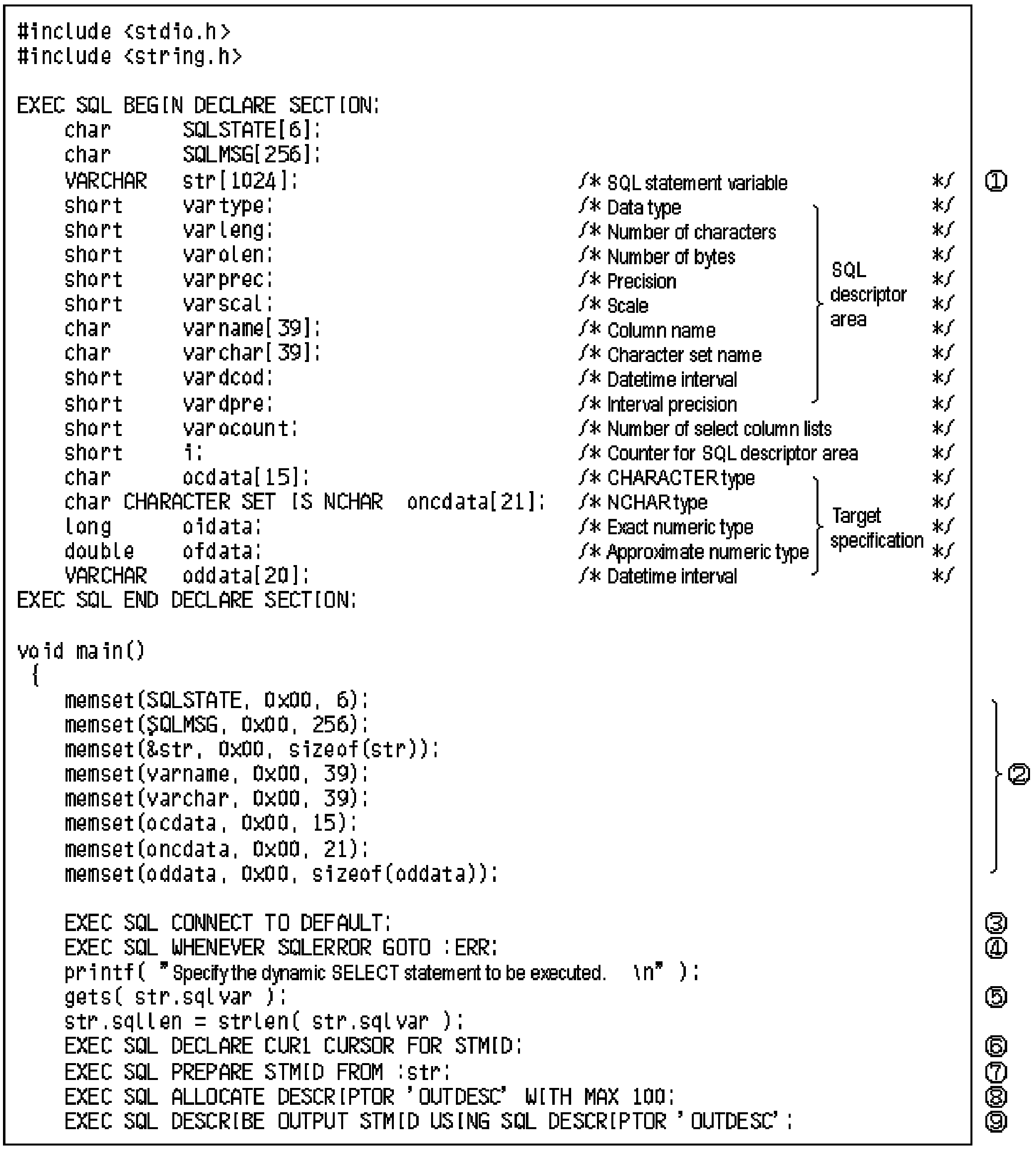
To fetch data continuously, prepare and execute a dynamic SELECT statement. An example of an application program that executes a dynamic SELECT statement using the SQL descriptor area is shown in "Figure: Example of application program that executes a dynamic SELECT statement."

Example:

This is an example of a dynamic SELECT statement entered from a terminal.

[Figure: Example of application program that executes a dynamic SELECT statement]

(Continued)



```
#include <stdio.h>
#include <string.h>

EXEC SQL BEGIN DECLARE SECTION;
    char      SQLSTATE[6];
    char      SQLMSG[256];
    VARCHAR   str[1024];                     /* SQL statement variable              */   ①
    short     vartype;                       /* Data type                           */
    short     varleng;                       /* Number of characters                */
    short     varolen;                       /* Number of bytes          SQL        */
    short     varprec;                       /* Precision                descriptor */
    short     varscal;                       /* Scale                    area       */
    char      varname[39];                   /* Column name                         */
    char      varchar[39];                   /* Character set name                  */
    short     vardcod;                       /* Datetime interval                   */
    short     vardpre;                       /* Interval precision                  */
    short     varocount;                     /* Number of select column lists       */
    short     i;                             /* Counter for SQL descriptor area     */
    char      ocdata[15];                    /* CHARACTER type                      */
    char CHARACTER SET IS NCHAR  oncdata[21]; /* NCHAR type            Target       */
    long      oidata;                        /* Exact numeric type     specification */
    double    ofdata;                        /* Approximate numeric type            */
    VARCHAR   oddata[20];                    /* Datetime interval                   */
EXEC SQL END DECLARE SECTION;

void main()
 {
    memset(SQLSTATE, 0x00, 6);
    memset($QLMSG, 0x00, 256);
    memset(&str, 0x00, sizeof(str));
    memset(varname, 0x00, 39);                                                           ②
    memset(varchar, 0x00, 39);
    memset(ocdata, 0x00, 15);
    memset(oncdata, 0x00, 21);
    memset(oddata, 0x00, sizeof(oddata));

    EXEC SQL CONNECT TO DEFAULT;                                                         ③
    EXEC SQL WHENEVER SQLERROR GOTO :ERR;                                                ④
    printf( "Specify the dynamic SELECT statement to be executed.   \n" );
    gets( str.sqlvar );                                                                  ⑤
    str.sqllen = strlen( str.sqlvar );
    EXEC SQL DECLARE CUR1 CURSOR FOR STMID;                                              ⑥
    EXEC SQL PREPARE STMID FROM :str;                                                    ⑦
    EXEC SQL ALLOCATE DESCRIPTOR 'OUTDESC' WITH MAX 100;                                 ⑧
    EXEC SQL DESCRIBE OUTPUT STMID USING SQL DESCRIPTOR 'OUTDESC';                       ⑨
```

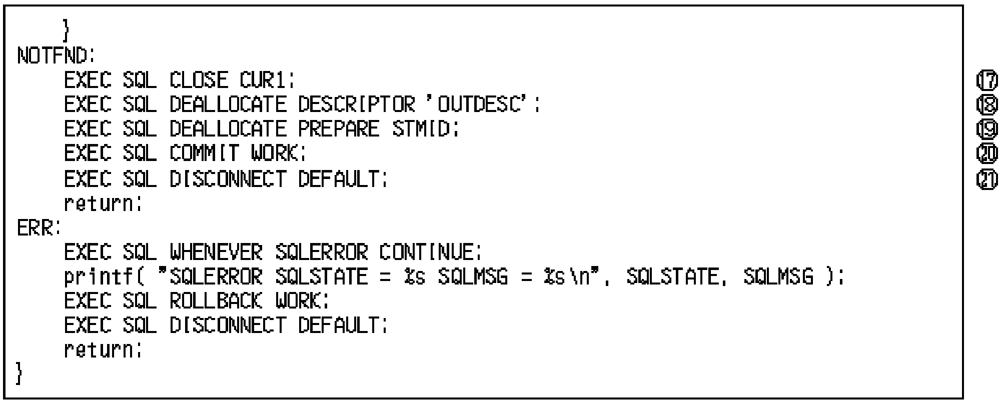
(Continued)

```
/****************/
/* Select column list */
/****************/
EXEC SQL GET DESCRIPTOR 'OUTDESC' :varocount = COUNT;          ⑩
for( i = 1; i <= varocount; i++ ) {
    EXEC SQL GET DESCRIPTOR 'OUTDESC' VALUE :i
                                 :vartype = TYPE,
                                 :varleng = LENGTH,
                                 :varolen = OCTET_LENGTH,
                                 :varprec = PRECISION,          ⑪
                                 :varscal = SCALE,
                                 :varchar = CHARACTER_SET_NAME,
                                 :vardcod = DATETIME_INTERVAL_CODE,
                                 :vardpre = DATETIME_INTERVAL_PRECISION;
    /* CHAR type or NCHAR type */
    if( vartype == 1 || vartype == 12 ) {
        /* Character string type  */
        if( strncmp( varchar, "BASIC", 5 ) == 0 ) {
            EXEC SQL SET DESCRIPTOR 'OUTDESC' VALUE :i
                                          TYPE = 1,
                                          LENGTH = 14,
                                          OCTET_LENGTH = 14,
                                          CHARACTER_SET_NAME = 'BASIC';
        }
        /* National character string type   */
        else {
            EXEC SQL SET DESCRIPTOR 'OUTDESC' VALUE :i
                                          TYPE = 1,
                                          LENGTH = 10,
                                          OCTET_LENGTH = 20,
                                          CHARACTER_SET_NAME = 'NCHAR';
        }
    }
    /* INTEGER type or SMALLINT type     */
    else if( vartype == 4 || vartype == 5 ) {                   ⑫
        EXEC SQL SET DESCRIPTOR 'OUTDESC' VALUE :i
                                      TYPE = 4,
                                      PRECISION = 31,
                                      SCALE = 0;
    }
    /* NUMERIC type, DECIMAL type, REAL type, or DOUBLE PRECISION type   */
    else if( vartype == 2 || vartype == 3 ||
             vartype == 7 || vartype == 8 ) {
        EXEC SQL SET DESCRIPTOR 'OUTDESC' VALUE :i
                                      TYPE = 8,
                                      PRECISION = 52,
                                      SCALE = 0;
    }
```

(Continued)

```
        /* Datetime type or INTERVAL type  */
        else if( vartype == 9 || vartype == 10 ) {
            EXEC SQL SET DESCRIPTOR 'OUTDESC' VALUE :i
                                              TYPE = 12,
                                              LENGTH = 19,
                                              OCTET_LENGTH = 19,
                                              CHARACTER_SET_NAME = 'BASIC';
        }
    }
    EXEC SQL OPEN CUR1;                                                           ⑬
    EXEC SQL WHENEVER NOT FOUND GOTO :NOTFND;                                     ⑭
    for(;;) {
        EXEC SQL FETCH CUR1 INTO SQL DESCRIPTOR 'OUTDESC';                        ⑮
        /*************/
        /* Execution result */
        /*************/
        for( i = 1; i <= varocount; i++ ) {
            EXEC SQL GET DESCRIPTOR 'OUTDESC' VALUE :i
                                              :vartype = TYPE,
                                              :varchar = CHARACTER_SET_NAME;
            /* CHAR type or NCHAR type */
            if( vartype == 1 ) {
                /* Character string type */
                if( strncmp( varchar, "BASIC", 5 ) == 0 ) {
                    EXEC SQL GET DESCRIPTOR 'OUTDESC' VALUE :i
                                                      :varname = NAME,
                                                      :ocdata = DATA;
                }
                /* National character string type      */
                else {
                    EXEC SQL GET DESCRIPTOR 'OUTDESC' VALUE :i
                                                     :varname = NAME,
                                                     :oncdata = DATA;
                }
            }                                                                       ⑯
            /* Exact numeric type     */
            else if( vartype == 4 ) {
                EXEC SQL GET DESCRIPTOR 'OUTDESC' VALUE :i
                                                  :varname = NAME,
                                                  :oidata = DATA;
            }
            /* Approximate numeric type      */
            else if( vartype == 8 ) {
                EXEC SQL GET DESCRIPTOR 'OUTDESC' VALUE :i
                                                  :varname = NAME,
                                                  :ofdata = DATA;
            }
            /* Datetime type or INTERVAL type    */
            else if( vartype == 12 ) {
                EXEC SQL GET DESCRIPTOR 'OUTDESC' VALUE :i
                                                  :varname = NAME,
                                                  :oddata = DATA;
                printf( "%s -> %s   ", varname, oddata.sqlvar );
            }
        }
```

(Continued)



```
    }
NOTFND:
    EXEC SQL CLOSE CUR1;                                        ⑰
    EXEC SQL DEALLOCATE DESCRIPTOR 'OUTDESC';                   ⑱
    EXEC SQL DEALLOCATE PREPARE STMID;                          ⑲
    EXEC SQL COMMIT WORK;                                       ⑳
    EXEC SQL DISCONNECT DEFAULT;                                ㉑
    return;
ERR:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    printf( "SQLERROR SQLSTATE = %s SQLMSG = %s\n", SQLSTATE, SQLMSG );
    EXEC SQL ROLLBACK WORK;
    EXEC SQL DISCONNECT DEFAULT;
    return;
}
```

① The SQL statement variable is the area in which an SQL statement is stored. A variable-length character string is expanded into the following structure when the program is compiled:

```
        struct {
            short  sqllen;          /* Length area variable            */
            char   sqlvar[1024];    /* Character string area variable  */
        }str;
```

② Initializes variables.
③ Opens a connection.
④ If an error other than "no data" occurs during execution of the SQL statement, the program branches to ERR, outputs error information, and terminates.
⑤ Reads the dynamic SELECT statement entered from the terminal and stores it in str.
⑥ Declares cursor CUR1. Specifies SQL statement identifier STMID as the cursor specification.
⑦ Prepares the SQL statement in str for execution.
⑧ Allocates SQL descriptor area OUTDESC for the target specification.
⑨ Fetches the select column list information for the entered dynamic SELECT statement to SQL descriptor area OUTDESC.
⑩ Gets the number of the select column lists (COUNT value) set in OUTDESC.
⑪ Gets the information (values of TYPE, LENGTH, OCTET_LENGTH, PRECISION, SCALE, and CHARACTER_SET_NAME) for each select column list set in OUTDESC.
⑫ Sets the data type of the target specification that fetches execution results to OUTDESC.
  - Changes CHARACTER(n) and CHARACTER VARYING(n) to CHARACTER(14). (n indicates the length of the character string.)
  - Changes NATIONAL CHARACTER(n) and NATIONAL CHARACTER VARYING(n) to NATIONAL CHARACTER(10). (n indicates the length of the character string.)
  - Changes INTEGER and SMALLINT to INTEGER.
  - Changes NUMERIC(p, q), DECIMAL(p, q), REAL, and DOUBLE PRECISION to DOUBLE PRECISION. (p indicates the precision and q indicates the scale.)
  - Changes DATE, TIME, TIMESTAMP, and INTERVAL to CHARACTER VARYING(19). (n indicates the length of the character string.)

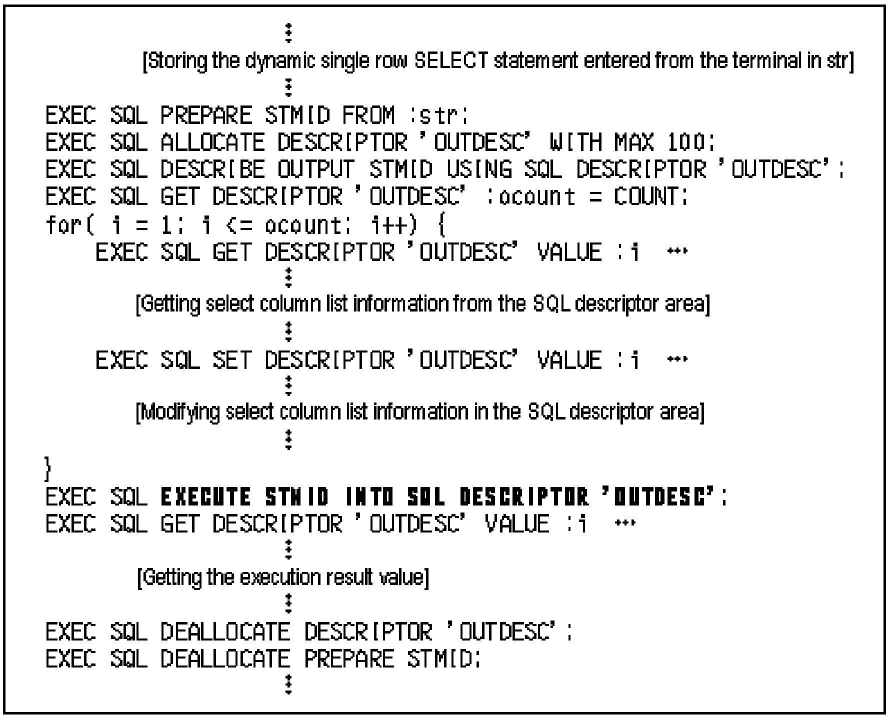
⑬ Opens cursor CUR1.
⑭ If search result data is not found, the program branches to NOTFND and terminates.
⑮ Positions cursor CUR1 and reads data. The execution result value is set in the DATA field of OUTDESC.
⑯ Fetches the NAME and DATA values from SQL descriptor area OUTDESC and stores them in host variables that match the data types in the target specification.
⑰ Closes cursor CUR1.
⑱ Deallocates SQL descriptor area OUTDESC.
⑲ Deallocates the prepared statement corresponding to SQL identifier STMID.
⑳ Terminates the current transaction.
㉑ Closes the connection.

7.2.2.2 Preparing and executing a dynamic single row SELECT statement (for SQL descriptor area)

To fetch data from one row, prepare and execute a dynamic single row SELECT statement. To execute a dynamic single row SELECT statement, use an EXECUTE statement in which the USING clause is specified. An example of a dynamic single row SELECT statement using the SQL descriptor area is shown below. For information about the SQL statements that can use the SQL descriptor area, see the examples in "Figure: Procedure for fetching execution results for a prepared statement."

Example:

This is an example of a dynamic single row SELECT statement entered from a terminal.



```
                    ⋮
          [Storing the dynamic single row SELECT statement entered from the terminal in str]
                    ⋮
EXEC SQL PREPARE STMID FROM :str;
EXEC SQL ALLOCATE DESCRIPTOR 'OUTDESC' WITH MAX 100;
EXEC SQL DESCRIBE OUTPUT STMID USING SQL DESCRIPTOR 'OUTDESC';
EXEC SQL GET DESCRIPTOR 'OUTDESC' :ocount = COUNT;
for( i = 1; i <= ocount; i++) {
    EXEC SQL GET DESCRIPTOR 'OUTDESC' VALUE :i  ⋯
                    ⋮
          [Getting select column list information from the SQL descriptor area]
                    ⋮
    EXEC SQL SET DESCRIPTOR 'OUTDESC' VALUE :i  ⋯
                    ⋮
          [Modifying select column list information in the SQL descriptor area]
                    ⋮
}
EXEC SQL EXECUTE STMID INTO SQL DESCRIPTOR 'OUTDESC';
EXEC SQL GET DESCRIPTOR 'OUTDESC' VALUE :i  ⋯
                    ⋮
          [Getting the execution result value]
                    ⋮
EXEC SQL DEALLOCATE DESCRIPTOR 'OUTDESC';
EXEC SQL DEALLOCATE PREPARE STMID;
                    ⋮
```

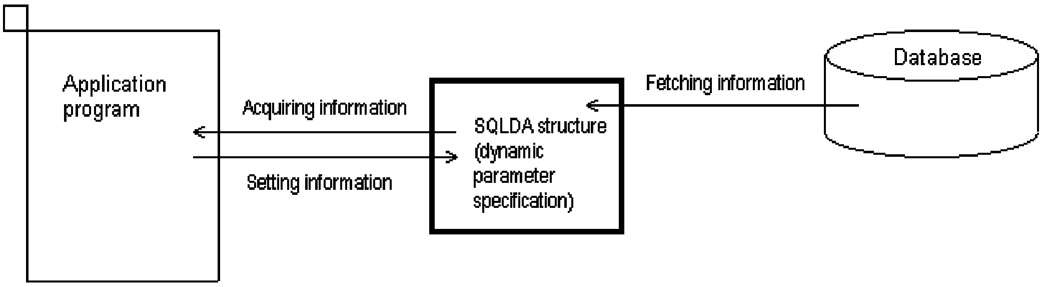
7.3 Dynamically Modifying and Executing SQL Statement Conditions

This section describes the following methods that can be used to dynamically modify and execute SQL statement conditions.

- Manipulating the SQLDA structure
    - Preparing and executing dynamic SELECT statements (for SQLDA structure)
    - Preparing and executing dynamic single-row SELECT statements (for SQLDA structure)
- Manipulating the SQL descriptor area
    - Preparing and executing dynamic SELECT statements (for SQL descriptor area)
    - Preparing and executing dynamic single-row SELECT statements (for SQL descriptor area)
- Executing other prepared statements
- Executing prepared statements for which variable attributes are known

7.3.1 Manipulating the SQLDA structure

To manipulate data after only dynamically modifying the SQL statement conditions, use dynamic parameter specifications. Store the values to be input in host variables. When the value of a host variable is set, the dynamic parameter specification information must match the attributes of the host variable. Fetch dynamic parameter specification information from the database to the SQLDA structure, modify the attributes, and specify the value. The following figure shows the position of the SQLDA structure.


Application program
Acquiring information
Setting information
SQLDA structure (dynamic parameter specification)
Fetching information
Database

The SQLDA structure holds dynamic parameter specification information.

The SQLDA structure consists of item descriptor area SQLVAR, the maximum number of SQLVAR elements SQLN, and the number of effective elements SQLD.

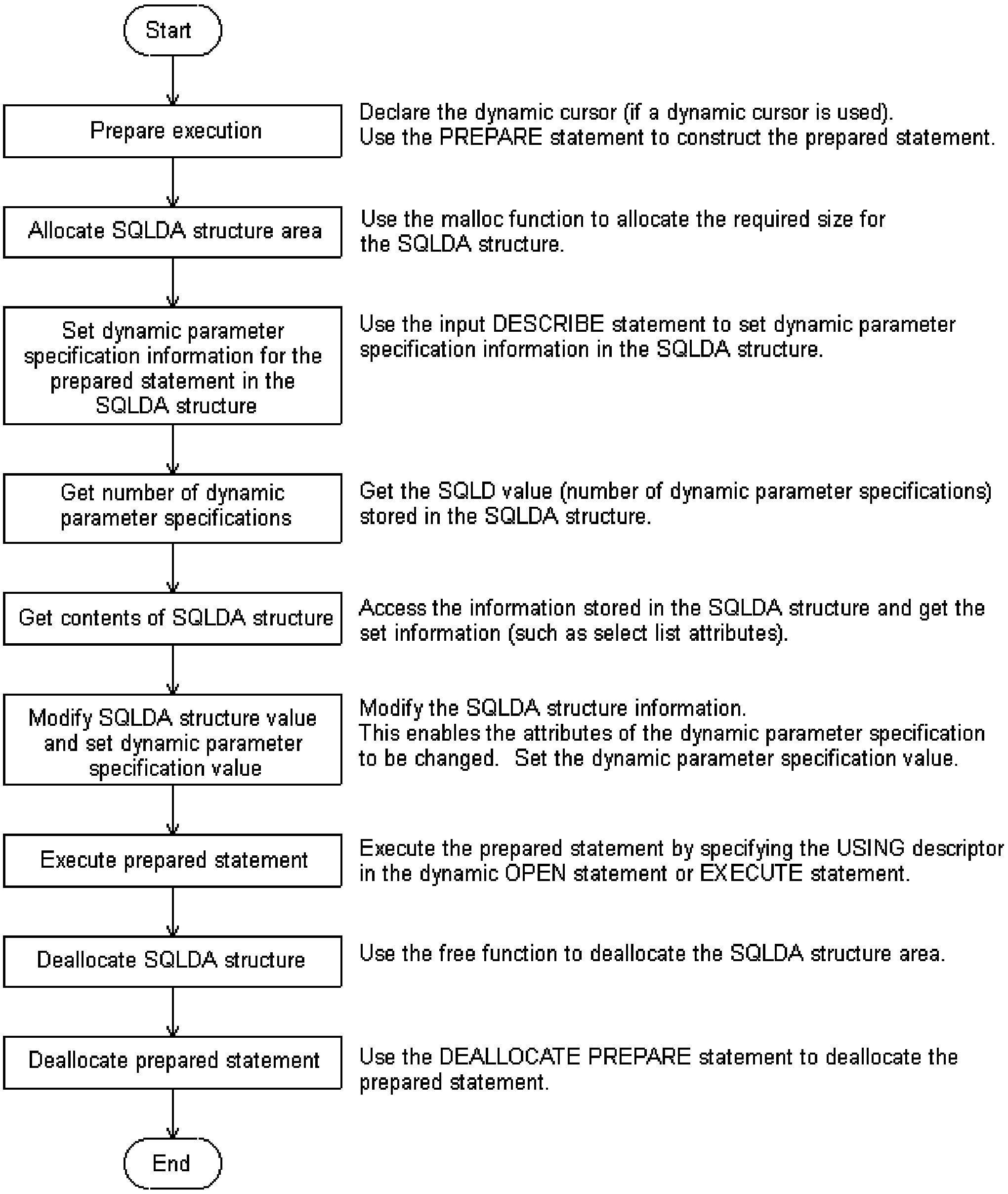
Item descriptor area SQLVAR consists of arrays in which dynamic parameter specification information is stored. SQLD indicates the number of dynamic parameter specifications in the SQLDA structure. The data type is exact numeric with binary precision.

For more information about the SQLDA structure, see 7.2.1 Preparing SQL statements and manipulating the SQLDA structure."

■Procedure for setting dynamic parameter specification values

"Figure: Procedure for setting dynamic parameter specification values for prepared statements" shows how to use the USING descriptor to set dynamic parameter specification values.

[Figure: Procedure for setting dynamic parameter specification values for prepared statements]


Start
Prepare execution
Allocate SQLDA structure area
Set dynamic parameter specification information for the prepared statement in the SQLDA structure
Get number of dynamic parameter specifications
Get contents of SQLDA structure
Modify SQLDA structure value and set dynamic parameter specification value
Execute prepared statement
Deallocate SQLDA structure
Deallocate prepared statement
End

Declare the dynamic cursor (if a dynamic cursor is used). Use the PREPARE statement to construct the prepared statement.

Use the malloc function to allocate the required size for the SQLDA structure.

Use the input DESCRIBE statement to set dynamic parameter specification information in the SQLDA structure.

Get the SQLD value (number of dynamic parameter specifications) stored in the SQLDA structure.

Access the information stored in the SQLDA structure and get the set information (such as select list attributes).

Modify the SQLDA structure information. This enables the attributes of the dynamic parameter specification to be changed. Set the dynamic parameter specification value.

Execute the prepared statement by specifying the USING descriptor in the dynamic OPEN statement or EXECUTE statement.

Use the free function to deallocate the SQLDA structure area.

Use the DEALLOCATE PREPARE statement to deallocate the prepared statement.

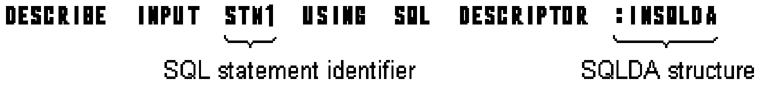
■SQL statements used to manipulate the SQLDA structure

As shown in "Figure: Procedure for setting dynamic parameter specification values for prepared statements," use SQL statements to fetch dynamic parameter specification information from the database to the SQLDA structure and to get dynamic parameter specification information from the SQLDA structure. The SQL statements used to manipulate the SQLDA structure and their functions are explained below.

◆DESCRIBE statement

The DESCRIBE statement fetches the dynamic parameter specification information for the prepared statement prepared by the PREPARE statement in the SQLDA structure. This type of statement is called an input DESCRIBE statement.

An example of an input DESCRIBE statement that fetches dynamic parameter specification information for the prepared statement corresponding to SQL statement identifier STM1 to the SQLDA structure is shown below.

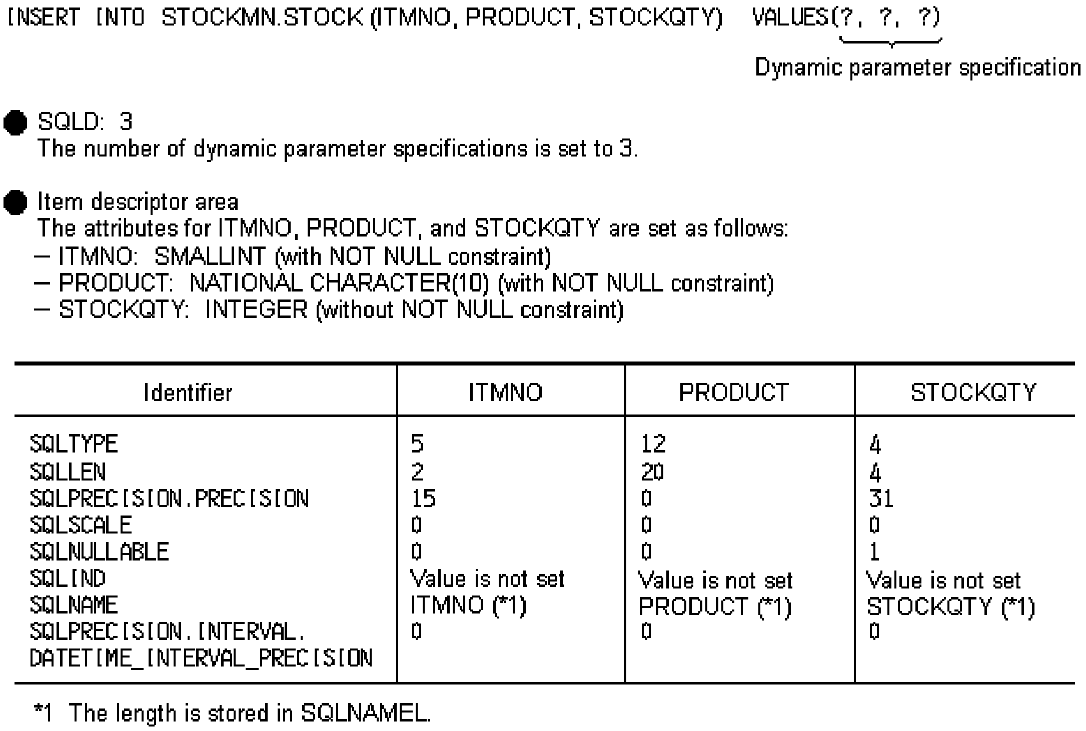


```
DESCRIBE INPUT STM1 USING SQL DESCRIPTOR :INSQLDA
```

SQL statement identifier

SQLDA structure

When the prepared statement corresponding to SQL statement identifier STM1 is as follows, the SQLDA structure variable will have the following contents:



```
INSERT INTO STOCKMN.STOCK (ITMNO, PRODUCT, STOCKQTY) VALUES(?, ?, ?)
```

Dynamic parameter specification

- SQLD: 3
  The number of dynamic parameter specifications is set to 3.
- Item descriptor area
  The attributes for ITMNO, PRODUCT, and STOCKQTY are set as follows:
  - ITMNO: SMALLINT (with NOT NULL constraint)
  - PRODUCT: NATIONAL CHARACTER(10) (with NOT NULL constraint)
  - STOCKQTY: INTEGER (without NOT NULL constraint)

Identifier	ITMNO	PRODUCT	STOCKQTY
SQLTYPE	5	12	4
SQLLEN	2	20	4
SQLPRECISION.PRECISION	15	0	31
SQLSCALE	0	0	0
SQLNULLABLE	0	0	1
SQLIND	Value is not set	Value is not set	Value is not set
SQLNAME	ITMNO (*1)	PRODUCT (*1)	STOCKQTY (*1)
SQLPRECISION.INTERVAL.DATETIME_INTERVAL_PRECISION	0	0	0

*1 The length is stored in SQLNAMEL.

This section describes only the SQL statements used to manipulate a SQLDA structure containing dynamic parameter specification information. For information about other SQL statements, see "7.2.1 Preparing SQL statements and manipulating the SQLDA structure."

7.3.1.1 Preparing and executing dynamic SELECT statements (for SQLDA structure)

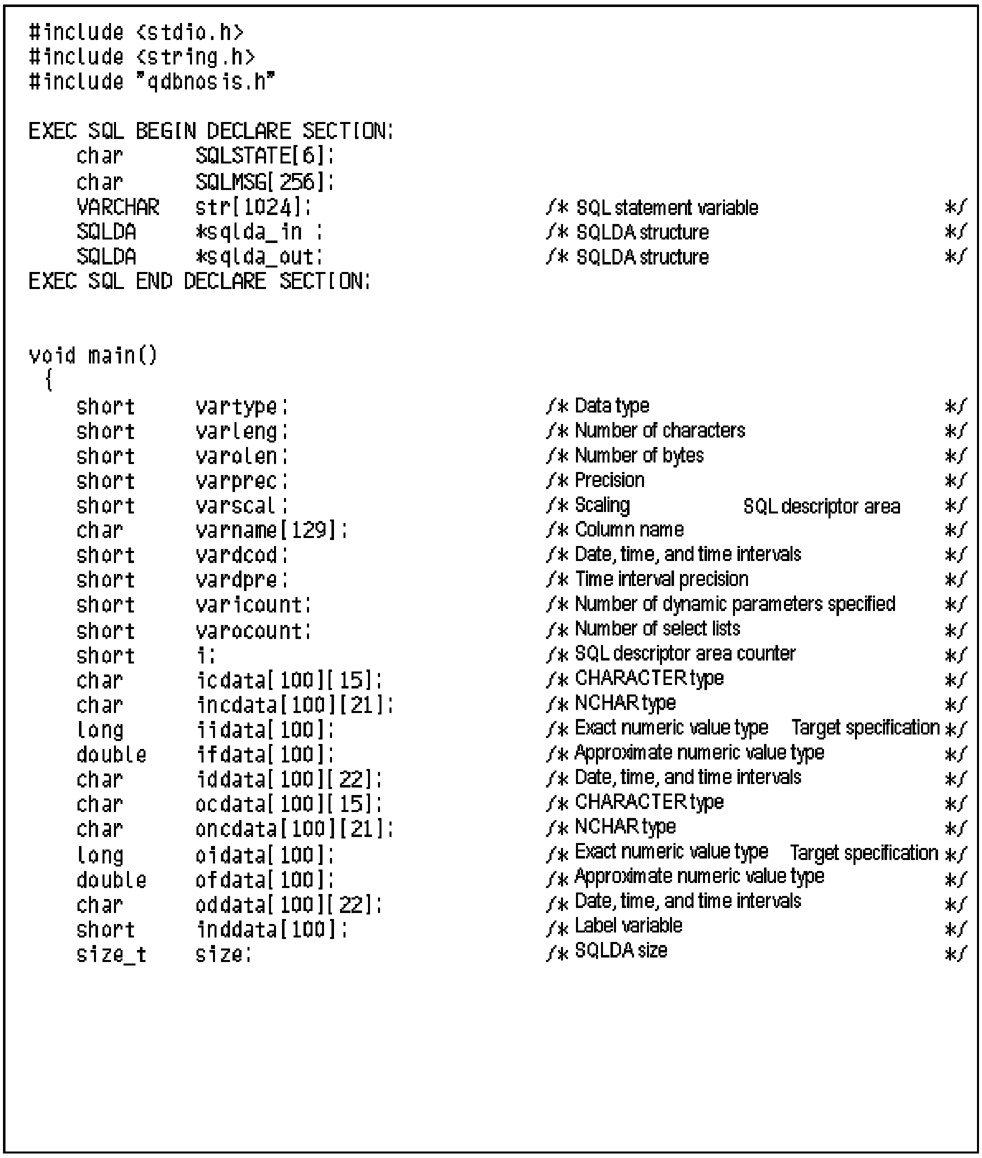
To fetch data continuously, prepare and execute a dynamic SELECT statement. "Figure: Example of an application program that uses the SQLDA structure (with dynamic parameter specification)" shows an example of an application program that uses the SQLDA structure to execute a dynamic SELECT statement with the dynamic parameter specification.

Example:

> This is an example of a dynamic SELECT statement entered from a terminal. Data is searched using the value of the dynamic parameter specification entered from the terminal as the search condition, and the value is displayed.

[Figure: Example of an application program that uses the SQLDA structure (with dynamic parameter specification)]

(Continued)



```
#include <stdio.h>
#include <string.h>
#include "qdbnosis.h"

EXEC SQL BEGIN DECLARE SECTION;
    char       SQLSTATE[6];
    char       SQLMSG[256];
    VARCHAR    str[1024];                  /* SQL statement variable                   */   ①
    SQLDA      *sqlda_in ;                 /* SQLDA structure                          */
    SQLDA      *sqlda_out;                 /* SQLDA structure                          */
EXEC SQL END DECLARE SECTION;


void main()
 {
    short      vartype;                    /* Data type                                */
    short      varleng;                    /* Number of characters                     */
    short      varolen;                    /* Number of bytes                          */
    short      varprec;                    /* Precision                                */
    short      varscal;                    /* Scaling            SQL descriptor area   */
    char       varname[129];               /* Column name                              */
    short      vardcod;                    /* Date, time, and time intervals           */
    short      vardpre;                    /* Time interval precision                  */
    short      varicount;                  /* Number of dynamic parameters specified   */
    short      varocount;                  /* Number of select lists                   */
    short      i;                          /* SQL descriptor area counter              */
    char       icdata[100][15];            /* CHARACTER type                           */
    char       incdata[100][21];           /* NCHAR type                               */
    long       iidata[100];                /* Exact numeric value type  Target specification */
    double     ifdata[100];                /* Approximate numeric value type           */
    char       iddata[100][22];            /* Date, time, and time intervals           */
    char       ocdata[100][15];            /* CHARACTER type                           */
    char       oncdata[100][21];           /* NCHAR type                               */
    long       oidata[100];                /* Exact numeric value type  Target specification */
    double     ofdata[100];                /* Approximate numeric value type           */
    char       oddata[100][22];            /* Date, time, and time intervals           */
    short      inddata[100];               /* Label variable                           */
    size_t     size;                       /* SQLDA size                               */
```

(Continued)

```
    memset(SQLSTATE, 0x00, 6);
    memset(SQLMSG, 0x00, 256);
    memset(&str, 0x00, sizeof(str));
    memset(varname, 0x00, sizeof(varname));
    memset(icdata, 0x00, sizeof(icdata));
    memset(incdata, 0x00, sizeof(incdata));                                  ②
    memset(iddata, 0x00, sizeof(iddata));
    memset(ocdata, 0x00, sizeof(ocdata));
    memset(oncdata, 0x00, sizeof(oncdata));
    memset(oddata, 0x00, sizeof(oddata));


    size = sizeof(SQLDA) + sizeof(sqlvar) * (100 - 1);                       ③
    sqlda_in = (SQLDA *)malloc(size);
    sqlda_out = (SQLDA *)malloc(size);
    memset( sqlda_in, 0x00, size );
    memset( sqlda_out, 0x00, size );
    sqlda_in->SQLN = 100;                                                    ④
    sqlda_out->SQLN = 100;
    EXEC SQL CONNECT TO DEFAULT;                                             ⑤
    EXEC SQL WHENEVER SQLERROR GOTO :ERR;                                    ⑥
    printf( "Specify the dynamic SELECT statement to be executed   \n" );
    gets( str.sqlvar );                                                      ⑦
    str.sqllen = strlen( str.sqlvar );
    EXEC SQL DECLARE CUR1 CURSOR FOR STMID;                                  ⑧
EXEC SQL PREPARE STMID FROM :str;                                            ⑨
EXEC SQL DESCRIBE INPUT STMID USING SQL DESCRIPTOR :sqlda_in;                ⑩
EXEC SQL DESCRIBE OUTPUT STMID USING SQL DESCRIPTOR :sqlda_out;              ⑪
```

(Continued)

```
    /**************************/
    /* Dynamic parameter specification */
    /**************************/
    varicount = sqlda_in->SQLD;                                                  ⑫
for( i = 1; i <= varicount; i++ ) {
       vartype = sqlda_in->SQLVAR[i].SQLTYPE;
       varleng = sqlda_in->SQLVAR[i].SQLLEN;
       varprec = sqlda_in->SQLVAR[i].SQLPRECISION.PRECISION;                     ⑬
       varscal = sqlda_in->SQLVAR[i].SQLSCALE;
       vardcod = sqlda_in->SQLVAR[i].SQLSCALE;
       vardpre = sqlda_in->SQLVAR[i].SQLPRECISION.INTERVAL.
       DATETIME_INTERVAL_PRECISION;
       memcpy( varname, sqlda_out->SQLVAR[i].SQLNAME.SQLNAMEC,
                         sqlda_out->SQLVAR[i].SQLNAME.SQLNAMEL );
       varname[sqlda_out->SQLVAR[i].SQLNAME.SQLNAMEL] = 0x00;
         /* CHAR type    */
        if( vartype == 1 || vartype == 12 ) {
            printf( "Specify the value for %s \n", varname );
            scanf( "%s", &icdata[i][0] );
            sqlda_in->SQLVAR[i].SQLTYPE = 12;
            sqlda_in->SQLVAR[i].SQLLEN = strlen( &icdata[i][0] );
            sqlda_in->SQLVAR[i].SQLDATA = (char *)&icdata[i][0];
        }
        else if( vartype == 11 || vartype == 13 ) {
            printf( " Specify the value for %s       \n", varname );
            scanf( "%s", &incdata[i][0] );
            sqlda_in->SQLVAR[i].SQLTYPE = 13;
            sqlda_in->SQLVAR[i].SQLLEN = strlen( &incdata[i][0] );
            sqlda_in->SQLVAR[i].SQLDATA = (char *)&incdata[i][0];
        }
        /* INTEGER type or SMALLINT type       */
        else if( vartype == 4 || vartype == 5 ) {                                ⑭
            printf( " Specify the value for %s     \n", varname );
            scanf( "%ld", &iidata[i] );
            sqlda_in->SQLVAR[i].SQLTYPE = 4;
            sqlda_in->SQLVAR[i].SQLLEN  = 4;
            sqlda_in->SQLVAR[i].SQLPRECISION.PRECISION = 31;
            sqlda_in->SQLVAR[i].SQLSCALE = 0;
            sqlda_in->SQLVAR[i].SQLDATA = (char *)&iidata[i];
        }
```

(Continued)

```
       /* NUMERIC type, DECIMAL type, REAL type, or DOUBLE PRECISION type   */
        else if( vartype == 2 || vartype == 3 ||
                 vartype == 7 || vartype == 8 ) {
            printf( " Specify the value for %s     \n", varname );
            scanf( "%lf", &ifdata[i] );
            sqlda_in->SQLVAR[i].SQLTYPE = 8;
            sqlda_in->SQLVAR[i].SQLLEN  = 8;
            sqlda_in->SQLVAR[i].SQLPRECISION.PRECISION = 52;
            sqlda_in->SQLVAR[i].SQLSCALE = 0;
            sqlda_in->SQLVAR[i].SQLDATA = (char *)&ifdata[i];
        }
        /* Datetime type or INTERVAL type   */
        else if( vartype == 9 || vartype == 10 ) {
            printf( " Specify the value for %s      \n", varname );
            scanf( "%s", &iddata[i][0] );
            sqlda_in->SQLVAR[i].SQLTYPE = 12;
            sqlda_in->SQLVAR[i].SQLLEN = strlen( &iddata[i][0] );
            sqlda_in->SQLVAR[i].SQLDATA = (char *)&iddata[i][0];
        }
    }
    /**************/
    /*    Select list     */
    /**************/
    varocount = sqlda_out->SQLD;                                          ⑮
for( i = 0; i < varocount; i++ ) {
        vartype = sqlda_out->SQLVAR[i].SQLTYPE;
        varleng = sqlda_out->SQLVAR[i].SQLLEN;
        varprec = sqlda_out->SQLVAR[i].SQLPRECISION.PRECISION;             ⑯
        varscal = sqlda_out->SQLVAR[i].SQLSCALE;
    vardcod = sqlda_out->SQLVAR[i].SQLSCALE;
    vardpre = sqlda_out->SQLVAR[i].SQLPRECISION.INTERVAL.
              DATETIME_INTERVAL_PRECISION;
        /* CHAR type or VCHAR type */
        if( vartype == 1 || vartype == 12 ) {
            sqlda_out->SQLVAR[i].SQLTYPE = 1;
            sqlda_out->SQLVAR[i].SQLLEN = 14;
            sqlda_out->SQLVAR[i].SQLDATA = (char *)&ocdata[i][0];
        }
        /* NCHAR type or NVCHAR type    */
        else if( vartype == 11 || vartype == 13 ) {
            sqlda_out->SQLVAR[i].SQLTYPE = 11;                             ⑰
            sqlda_out->SQLVAR[i].SQLLEN = 20;
            sqlda_out->SQLVAR[i].SQLDATA = (char *)&oncdata[i][0];
        }
```

(Continued)

```
        /* INTEGER type or SMALLINT type   */
        else if( vartype == 4 || vartype == 5 ) {
            sqlda_out->SQLVAR[i].SQLTYPE = 4;
            sqlda_out->SQLVAR[i].SQLLEN  = 4;
            sqlda_out->SQLVAR[i].SQLPRECISION.PRECISION = 31;
            sqlda_out->SQLVAR[i].SQLSCALE = 0;
            sqlda_out->SQLVAR[i].SQLDATA = (char *)&oidata[i];
        }
        /* NUMERIC type, DECIMAL type, REAL type, or DOUBLE PRECISION type */
        else if( vartype == 2 || vartype == 3 ||
                 vartype == 7 || vartype == 8 ) {
            sqlda_out->SQLVAR[i].SQLTYPE = 8;
            sqlda_out->SQLVAR[i].SQLLEN  = 8;
            sqlda_out->SQLVAR[i].SQLPRECISION.PRECISION = 52;
            sqlda_out->SQLVAR[i].SQLSCALE = 0;
            sqlda_out->SQLVAR[i].SQLDATA = (char *)&ofdata[i];
        }
        /* Datetime type or INTERVAL type */
        else if( vartype == 9 || vartype == 10 ) {
            sqlda_out->SQLVAR[i].SQLTYPE = 12;
            sqlda_out->SQLVAR[i].SQLLEN = 21;
            sqlda_out->SQLVAR[i].SQLDATA = (char *)&oddata[i][0];
        }
    }
    EXEC SQL OPEN CUR1 USING SQL DESCRIPTOR :sqlda_in;                       ⑱
    EXEC SQL WHENEVER NOT FOUND GOTO :NOTFND;                                 ⑲
    for(;;) {
        EXEC SQL FETCH CUR1 INTO SQL DESCRIPTOR :sqlda_out;                   ⑳
        /***************/
        /* Execution result */
        /***************/
        for( i = 0; i < varocount; i++ ) {
            vartype = sqlda_out->SQLVAR[i].SQLTYPE;

            memcpy( varname, sqlda_out->SQLVAR[i].SQLNAME.SQLNAMEC,
                             sqlda_out->SQLVAR[i].SQLNAME.SQLNAMEL );
            varname[sqlda_out->SQLVAR[i].SQLNAME.SQLNAMEL] = 0x00;
            /* CHAR type */
            if( vartype == 1 ) {
                printf( "%s -> %s   ", varname, &ocdata[i][0] );
            }
            /* NCHAR type */
            else if( vartype == 11 ) {
                printf( "%s -> %s   ", varname, &oncdata[i][0] );
            }
```

(Continued)

```
            /* Exact numeric value type    */
            else if( vartype == 4 ) {
                printf( "%s -> %d   ", varname, oidata[i] );
            }
            /* Approximate numeric value type  */
            else if( vartype == 8 ) {
                printf( "%s -> %lf  ", varname, ofdata[i] );
            }
            /* Datetime type or INTERVAL type    */
            else if( vartype == 12 ) {
                printf( "%s -> %s   ", varname, &oddata[i][2] );
            }
        }
        printf( "\n" );
    }
NOTFND:
    EXEC SQL CLOSE CUR1;                                                    ㉑
     EXEC SQL DEALLOCATE PREPARE STMID;                                     ㉒
   EXEC SQL COMMIT WORK;                                                    ㉓
   EXEC SQL DISCONNECT DEFAULT;                                             ㉔
   free( sqlda_in );
    free( sqlda_out );
    return;
ERR:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    printf( "SQLERROR SQLSTATE = %s SQLMSG = %s\n", SQLSTATE, SQLMSG );
    EXEC SQL ROLLBACK WORK;
    EXEC SQL DISCONNECT DEFAULT;
    free( sqlda_in );
    free( sqlda_out );
    return;
}
```

① The SQL statement variable is the area in which the SQL statements is stored. A variable-length character string is expanded into the following structure when the program is compiled:

```
        struct {
            short  sqllen;           /* Length area variable          */
            char   sqlvar[100];      /* Character string area variable */
        }str;
```

② Initializes host variables.
③ Allocates an SQLDA structure area.
④ Specifies the number of SQLVAR arrays (SQLN value).
⑤ Opens a connection.
⑥ If an error other than "no data" occurs during execution of the SQL statement, the program branches to ERR, outputs error information, and terminates.
⑦ Reads the dynamic SELECT statement entered from the terminal and stores it in str.
⑧ Declares cursor CUR1. Specifies SQL statement identifier STMID as the cursor specification.
⑨ Prepares the SQL statement in str for execution.
⑩ Dynamic parameter specification information for the input dynamic SELECT statement is fetched to the SQLDA structure (sqlda_in).
⑪ Fetches the select list information for the input dynamic SELECT statement to the SQLDA structure (sqlda_out).
⑫ Gets the number of select lists (SQLD value) set in the SQLDA structure (sqlda_in).
⑬ Gets the information for each dynamic parameter specification (SQLTYPE, SQLLEN, SQLPRECISION, SQLSCALE DATETIME_INTERVAL_CODE, DATETIME_INTERVAL_PRECISION values) set in the SQLDA structure (sqlda_in).
⑭ Modifies the data type specified in the dynamic parameter set in the SQLDA structure (sqlda_in), and sets the value of the dynamic parameter specification read from the terminal.
- Changes CHARACTER(n) and CHARACTER VARYING(n) to CHARACTER(14). (n indicates the length of the character string.)
- Changes NATIONAL CHARACTER(n) and NATIONAL CHARACTER VARYING(n) to NATIONAL CHARACTER(10). (n indicates the length of the character string.)
- Changes INTEGER and SMALLINT to INTEGER.
- Changes NUMERIC(p, q), DECIMAL(p, q), REAL, and DOUBLE PRECISION to DOUBLE PRECISION. (p indicates the precision and q indicates the scale.)
- Changes DATE, TIME, TIMESTAMP, and INTERVAL to CHARACTER VARYING(19). (n indicates the length of the character string.)

⑮ Gets the number of select lists (SQLD value) set in the SQLDA structure.
⑯ Gets the information for each select list (SQLTYPE, SQLLEN, SQLPRECISION, SQLSCALE values) set in the SQLDA structured.
⑰ Sets the data types and addresses of variables of the target specification from which the execution result is fetched to the SQLDA structure.
- Changes CHARACTER(n) and CHARACTER VARYING(n) to CHARACTER(14). (n indicates the length of the character string.)
- Changes NATIONAL CHARACTER(n) and NATIONAL CHARACTER VARYING(n) to NATIONAL CHARACTER(10). (n indicates the length of the character string.)
- Changes INTEGER and SMALLINT to INTEGER.
- Changes NUMERIC(p, q), DECIMAL(p, q), REAL, and DOUBLE PRECISION to DOUBLE PRECISION. (p indicates the precision and q indicates the scale.)
- Changes DATE, TIME, TIMESTAMP, and INTERVAL to CHARACTER VARYING(19). (n indicates the length of the character string.)

⑱ Opens cursor CUR1 using the value obtained in ⑭.
⑲ If search result data is not found, the program branches to NOTFND and terminates.
⑳ Positions cursor CUR1 and reads data. The execution result value is set at the address specified in SQLDATA of the SQLDA structure (sqlda_out).
㉑ Closes cursor CUR1.
㉒ Deallocates the prepared statement corresponding to SQL statement identifier STMID.
㉓ Terminates the current transaction.
㉔ Closes the connection

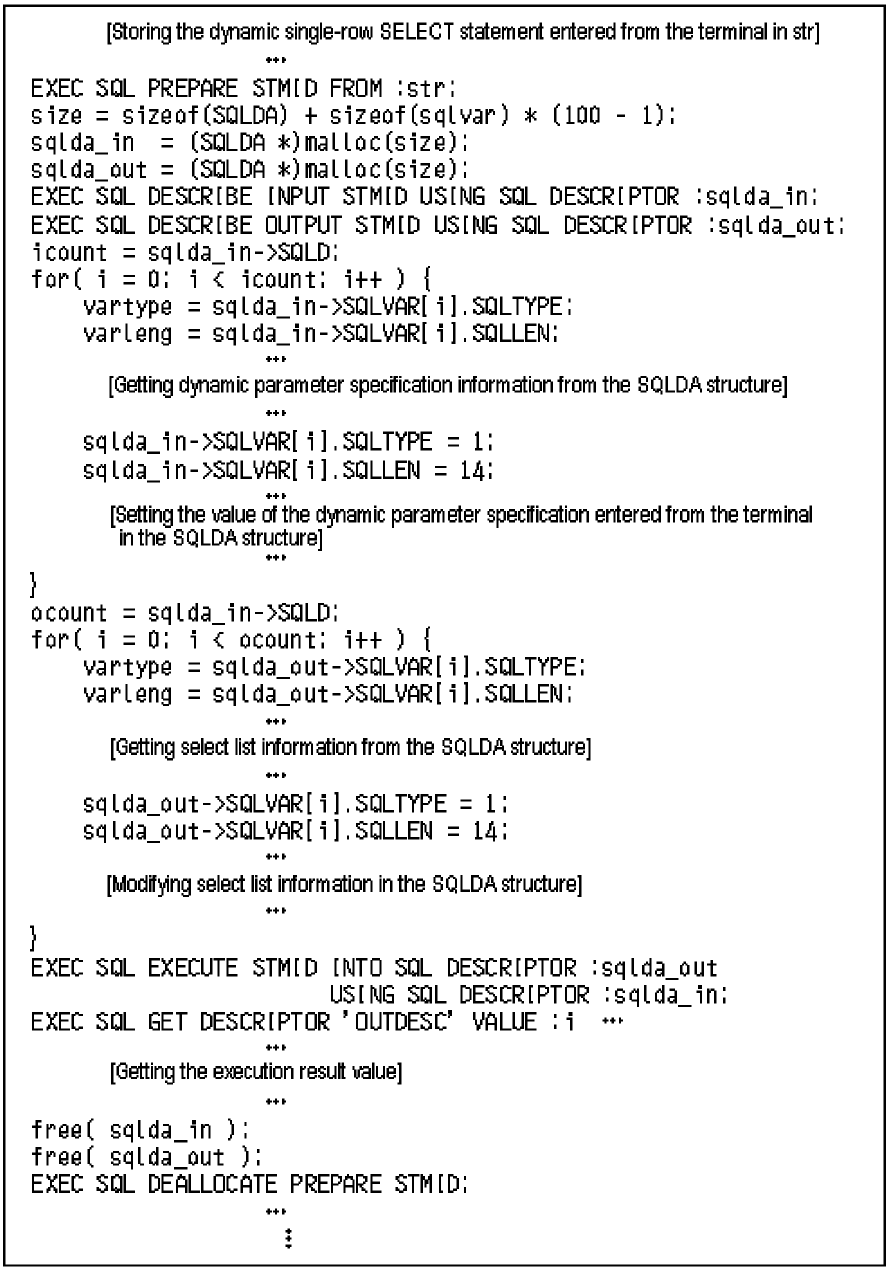
7.3.1.2 Preparing and executing dynamic single-row SELECT statements (for SQLDA structure)

To fetch single-row data, prepare and execute a dynamic single-row SELECT statement. To execute a dynamic single-row SELECT statement, use the EXECUTE statement with the USING clause specified.

An example of using the SQLDA structure to specify a dynamic single-row SELECT statement with the dynamic parameter specification is shown below. For information about the SQL statements used to manipulate the SQL structure, see "Figure: Procedure for setting dynamic parameter specification values for prepared statements."

Example:

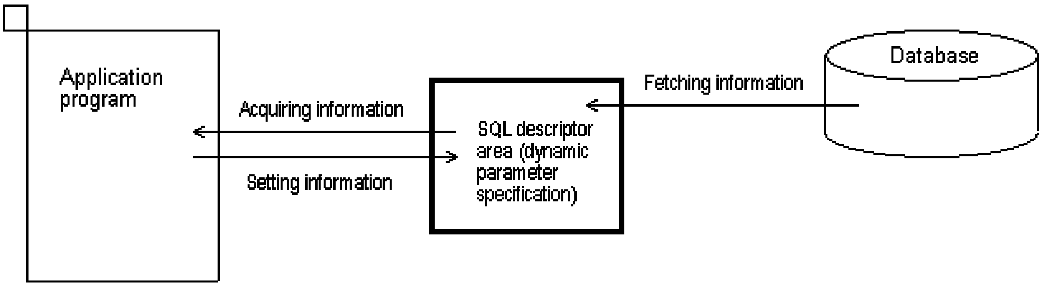
> This is an example of a dynamic single-row SELECT statement entered from a terminal. Data is searched using the value of the dynamic parameter specification entered from the terminal as the search condition, and the value is displayed.



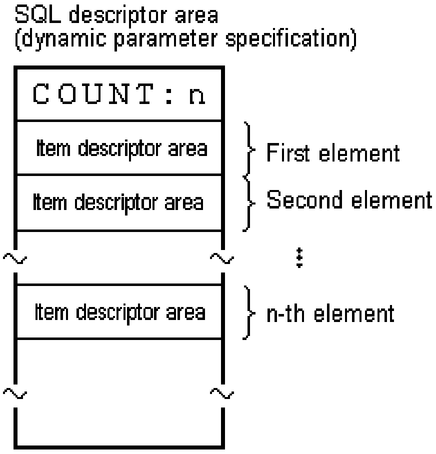
```
        [Storing the dynamic single-row SELECT statement entered from the terminal in str]
                      ...
EXEC SQL PREPARE STMID FROM :str;
size = sizeof(SQLDA) + sizeof(sqlvar) * (100 - 1);
sqlda_in  = (SQLDA *)malloc(size);
sqlda_out = (SQLDA *)malloc(size);
EXEC SQL DESCRIBE INPUT STMID USING SQL DESCRIPTOR :sqlda_in;
EXEC SQL DESCRIBE OUTPUT STMID USING SQL DESCRIPTOR :sqlda_out;
icount = sqlda_in->SQLD;
for( i = 0; i < icount; i++ ) {
    vartype = sqlda_in->SQLVAR[i].SQLTYPE;
    varleng = sqlda_in->SQLVAR[i].SQLLEN;
                      ...
        [Getting dynamic parameter specification information from the SQLDA structure]
                      ...
    sqlda_in->SQLVAR[i].SQLTYPE = 1;
    sqlda_in->SQLVAR[i].SQLLEN = 14;
                      ...
        [Setting the value of the dynamic parameter specification entered from the terminal
         in the SQLDA structure]
                      ...
}
ocount = sqlda_in->SQLD;
for( i = 0; i < ocount; i++ ) {
    vartype = sqlda_out->SQLVAR[i].SQLTYPE;
    varleng = sqlda_out->SQLVAR[i].SQLLEN;
                      ...
        [Getting select list information from the SQLDA structure]
                      ...
    sqlda_out->SQLVAR[i].SQLTYPE = 1;
    sqlda_out->SQLVAR[i].SQLLEN = 14;
                      ...
        [Modifying select list information in the SQLDA structure]
                      ...
}
EXEC SQL EXECUTE STMID INTO SQL DESCRIPTOR :sqlda_out
                       USING SQL DESCRIPTOR :sqlda_in;
EXEC SQL GET DESCRIPTOR 'OUTDESC' VALUE :i  ...
                      ...
        [Getting the execution result value]
                      ...
free( sqlda_in );
free( sqlda_out );
EXEC SQL DEALLOCATE PREPARE STMID;
                      ...
                       ⋮
```

7.3.2 Using the SQL descriptor area

To dynamically modify only the SQL statement condition and manipulate data, specify the dynamic parameter. Store values to be input in the host variable. When the host variable value is set, the dynamic parameter specification information must match the host variable attributes. Matching is accomplished by fetching dynamic parameter specification information from the database to the SQL descriptor area, modifying the attributes, and then specifying the value. The following figure shows the position of the SQL descriptor area.


Application program
Acquiring information
Setting information
SQL descriptor area (dynamic parameter specification)
Fetching information
Database

The SQL descriptor area either consists of only one identifier (COUNT) or consists of one identifier (COUNT) and one or more item descriptor areas (elements of the SQL descriptor area). The data type is exact numeric with binary precision. In the following figure, n indicates the number of dynamic parameter specifications in the SQL descriptor area.

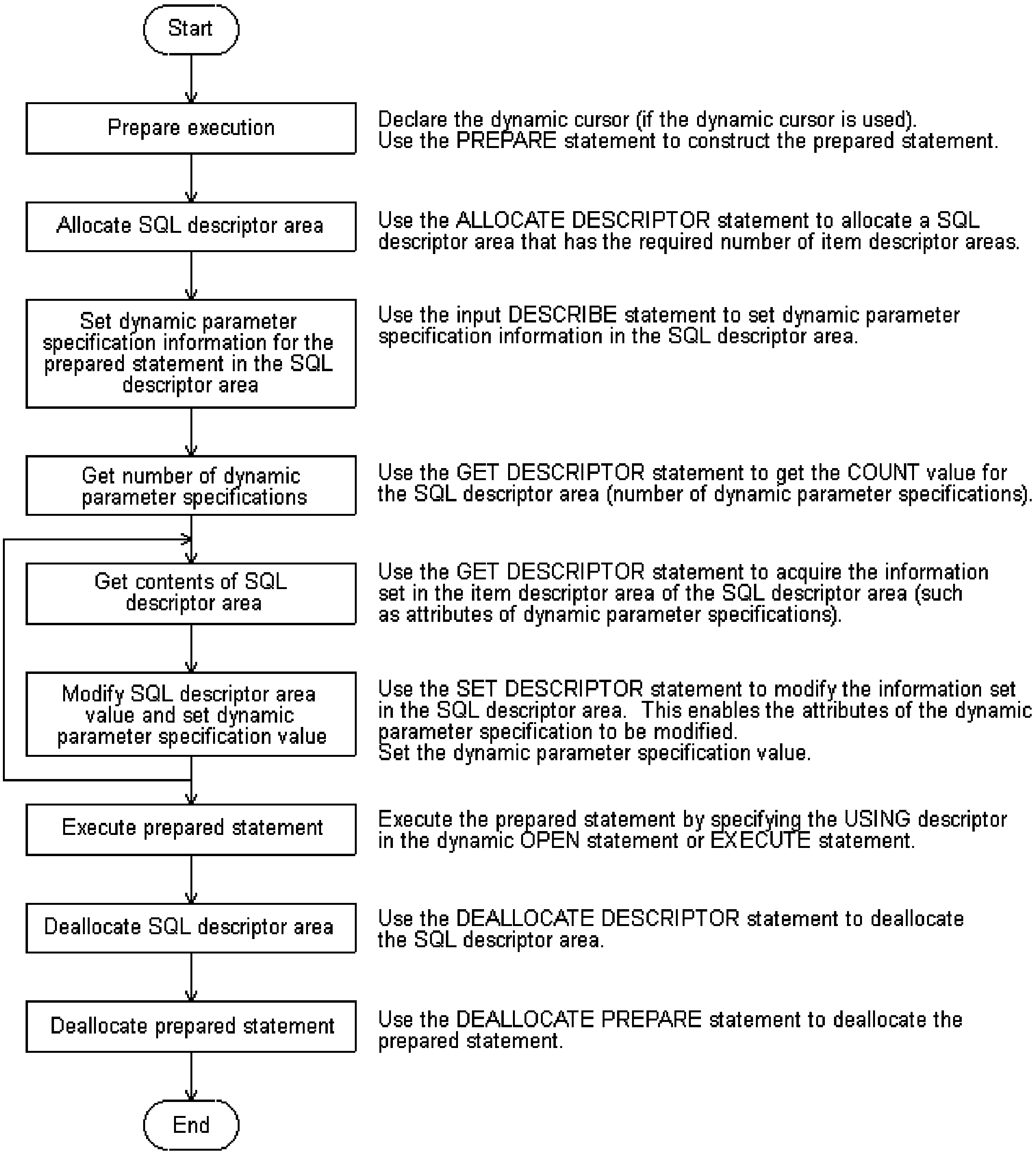

SQL descriptor area
(dynamic parameter specification)
COUNT: n
Item descriptor area
First element
Item descriptor area
Second element
Item descriptor area
n-th element

For more information about the SQL descriptor area, see "7.2.2 Preparing SQL statements and manipulating the SQL descriptor area."

■Procedure for setting dynamic parameter specification values

"Figure: Procedure for setting dynamic parameter specification values for prepared statements" shows how to use the USING descriptor to set dynamic parameter specification values.

[Figure: Procedure for setting dynamic parameter specification values for prepared statement s]


Start
Prepare execution
Allocate SQL descriptor area
Set dynamic parameter specification information for the prepared statement in the SQL descriptor area
Get number of dynamic parameter specifications
Get contents of SQL descriptor area
Modify SQL descriptor area value and set dynamic parameter specification value
Execute prepared statement
Deallocate SQL descriptor area
Deallocate prepared statement
End

Declare the dynamic cursor (if the dynamic cursor is used).
Use the PREPARE statement to construct the prepared statement.

Use the ALLOCATE DESCRIPTOR statement to allocate a SQL descriptor area that has the required number of item descriptor areas.

Use the input DESCRIBE statement to set dynamic parameter specification information in the SQL descriptor area.

Use the GET DESCRIPTOR statement to get the COUNT value for the SQL descriptor area (number of dynamic parameter specifications).

Use the GET DESCRIPTOR statement to acquire the information set in the item descriptor area of the SQL descriptor area (such as attributes of dynamic parameter specifications).

Use the SET DESCRIPTOR statement to modify the information set in the SQL descriptor area. This enables the attributes of the dynamic parameter specification to be modified.
Set the dynamic parameter specification value.

Execute the prepared statement by specifying the USING descriptor in the dynamic OPEN statement or EXECUTE statement.

Use the DEALLOCATE DESCRIPTOR statement to deallocate the SQL descriptor area.

Use the DEALLOCATE PREPARE statement to deallocate the prepared statement.

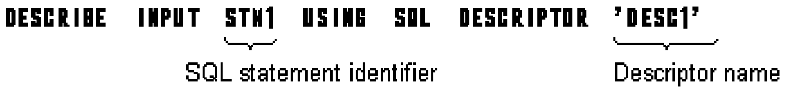
■SQL statements used to manipulate the SQL descriptor area

As shown in "Figure: Procedure for setting dynamic parameter specification values for prepared statements," use SQL statements to fetch dynamic parameter specification information from the database to the SQL descriptor area and get dynamic parameter specification information placed in the SQL descriptor area. The SQL statements used to manipulate the SQL descriptor area and their functions are described below.

◆DESCRIBE statement

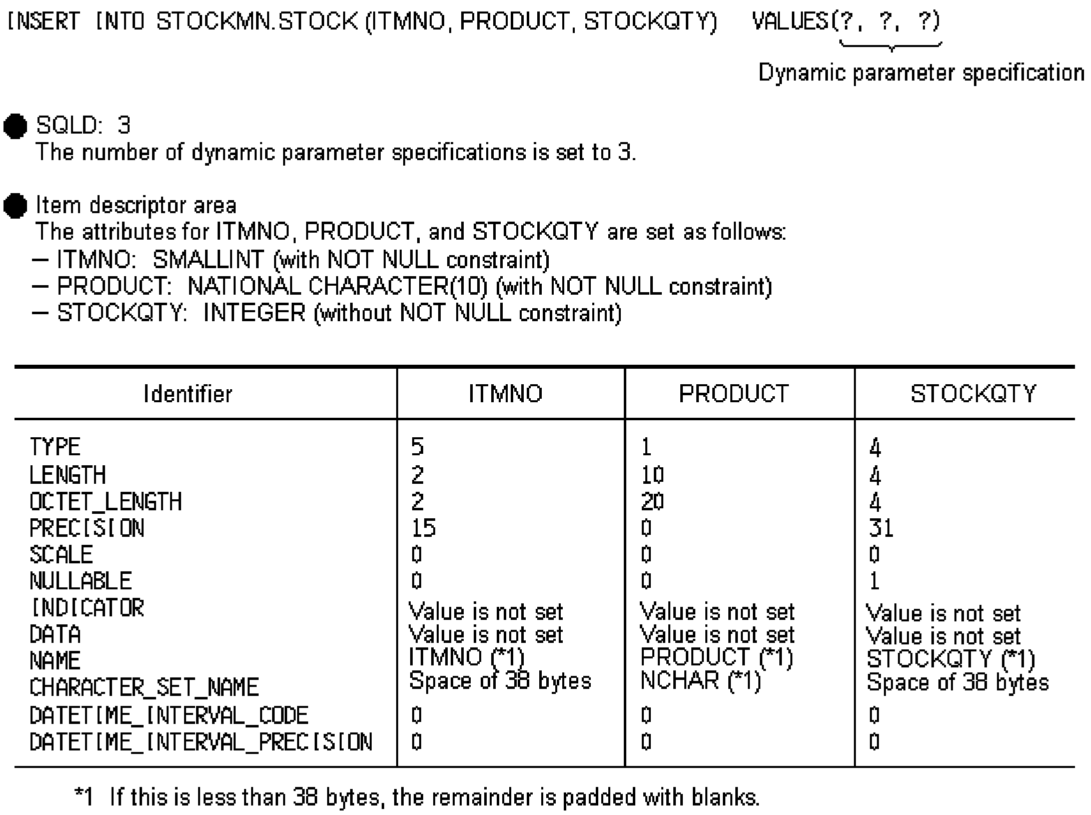
The DESCRIBE statement fetches the dynamic parameter specification information for the prepared statement prepared by the PREPARE statement in the SQL descriptor area. This type of statement is called an input DESCRIBE statement. When the descriptor name specified in the DESCRIBE statement is specified in the dynamic OPEN statement, the cursor must be closed.

An example of the input DESCRIBE statement that fetches dynamic parameter specification information for the prepared statement corresponding to SQL statement identifier STM1 to the SQL descriptor area is shown below.



```
DESCRIBE INPUT STM1 USING SQL DESCRIPTOR 'DESC1'
```

When the prepared statement corresponding to SQL statement identifier STM1 is as follows, SQL descriptor area DESC1 will have the following contents:



```
INSERT INTO STOCKMN.STOCK (ITMNO, PRODUCT, STOCKQTY) VALUES(?, ?, ?)
```

Dynamic parameter specification

- SQLD: 3
  The number of dynamic parameter specifications is set to 3.
- Item descriptor area
  The attributes for ITMNO, PRODUCT, and STOCKQTY are set as follows:
  - ITMNO: SMALLINT (with NOT NULL constraint)
  - PRODUCT: NATIONAL CHARACTER(10) (with NOT NULL constraint)
  - STOCKQTY: INTEGER (without NOT NULL constraint)

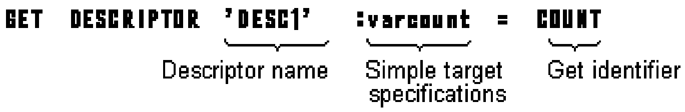
Identifier	ITMNO	PRODUCT	STOCKQTY
TYPE	5	1	4
LENGTH	2	10	4
OCTET_LENGTH	2	20	4
PRECISION	15	0	31
SCALE	0	0	0
NULLABLE	0	0	1
INDICATOR	Value is not set	Value is not set	Value is not set
DATA	Value is not set	Value is not set	Value is not set
NAME	ITMNO (*1)	PRODUCT (*1)	STOCKQTY (*1)
CHARACTER_SET_NAME	Space of 38 bytes	NCHAR (*1)	Space of 38 bytes
DATETIME_INTERVAL_CODE	0	0	0
DATETIME_INTERVAL_PRECISION	0	0	0

*1 If this is less than 38 bytes, the remainder is padded with blanks.

◆GET DESCRIPTOR statement

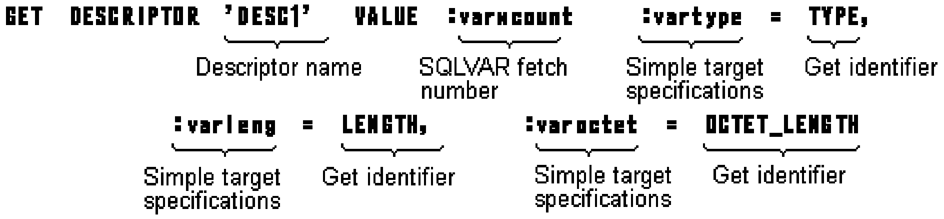
The GET DESCRIPTOR statement fetches information set in the SQL descriptor area to the host variable. The data type of the host variable must match the data type of the corresponding get identifier.

An example of fetching the number of dynamic parameter specifications to host variable varcount is shown below.



```
GET DESCRIPTOR 'DESC1' :varcount = COUNT
```

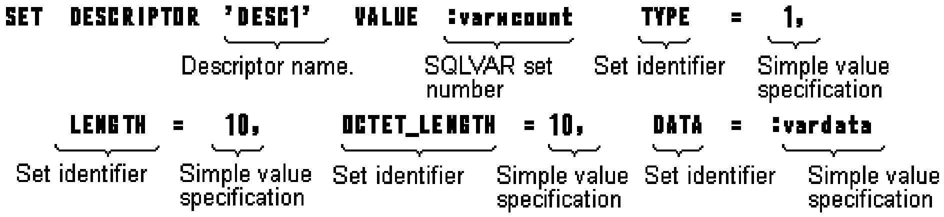
An example of fetching the contents of the dynamic parameter specification set in the SQL descriptor area to host variables vartype, varleng, and varoctet is shown below. The element number of the item descriptor area to be acquired should be the value set in varwcount.


SET DESCRIPTOR 'DESC1' VALUE :varwcount :vartype = TYPE,
Descriptor name
SQLVAR fetch number
Simple target specifications
Get identifier
:varleng = LENGTH, :varoctet = OCTET_LENGTH
Simple target specifications
Get identifier
Simple target specifications
Get identifier

◆**SET DESCRIPTOR statement**

The SET DESCRIPTOR statement sets the data type and value of the dynamic parameter specification in the SQL descriptor area. Specify them using the constant or host variable. The data type of the host variable must match the data type of each set identifier. When the descriptor name specified in the SET DESCRIPTOR statement is specified in a dynamic OPEN statement, the cursor must be closed.

An example of modifying the data type of the dynamic parameter specification in the SQL descriptor area and setting a value is shown below. In the example, the data type of host variable vardata in which the value to be set is stored is assume to be declared as CHARACTER(11). Change the data type of the dynamic parameter specification to CHARACTER(10). The element number of the item descriptor area to be set should be the value set in varwcount.


SET DESCRIPTOR 'DESC1' VALUE :varwcount TYPE = 1,
Descriptor name.
SQLVAR set number
Set identifier
Simple value specification
LENGTH = 10, OCTET_LENGTH = 10, DATA = :vardata
Set identifier
Simple value specification
Set identifier
Simple value specification
Set identifier
Simple value specification

This section describes only the SQL statements used to manipulate a SQL descriptor area containing dynamic parameter specification information. For information about other SQL statements, see "7.2.2 Preparing SQL statements and manipulating the SQL descriptor area."

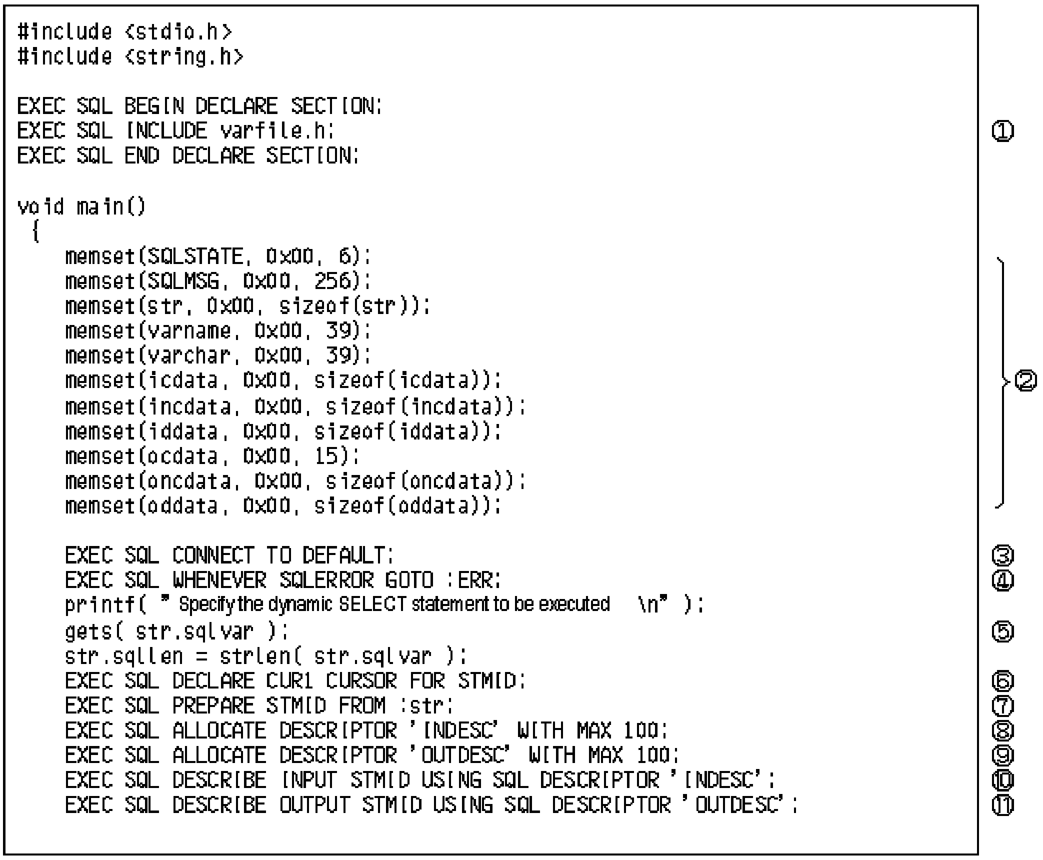
7.3.2.1 Preparing and executing dynamic SELECT statements (for SQL descriptor area)

To fetch data continuously, prepare and execute a dynamic SELECT statement. "Figure: Example of an application program that uses the SQL descriptor area (with dynamic parameter specification)" shows an example of an application program that uses the SQL descriptor area to execute a dynamic SELECT statement with the dynamic parameter specification.

Example:

> This is an example of a dynamic SELECT statement entered from a terminal. Data is searched using the value of the dynamic parameter specification entered from the terminal as the search condition, and the value is displayed.

[Figure: Example of an application program that uses the SQL descriptor area (with dynamic parameter specification)]



(Continued)

```
#include <stdio.h>
#include <string.h>

EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL INCLUDE varfile.h;                                              ①
EXEC SQL END DECLARE SECTION;

void main()
 {
    memset(SQLSTATE, 0x00, 6);
    memset(SQLMSG, 0x00, 256);
    memset(str, 0x00, sizeof(str));
    memset(varname, 0x00, 39);
    memset(varchar, 0x00, 39);
    memset(icdata, 0x00, sizeof(icdata));                                ②
    memset(incdata, 0x00, sizeof(incdata));
    memset(iddata, 0x00, sizeof(iddata));
    memset(ocdata, 0x00, 15);
    memset(oncdata, 0x00, sizeof(oncdata));
    memset(oddata, 0x00, sizeof(oddata));

    EXEC SQL CONNECT TO DEFAULT;                                         ③
    EXEC SQL WHENEVER SQLERROR GOTO :ERR;                                ④
    printf( " Specify the dynamic SELECT statement to be executed  \n" );
    gets( str.sqlvar );                                                  ⑤
    str.sqllen = strlen( str.sqlvar );
    EXEC SQL DECLARE CUR1 CURSOR FOR STMID;                              ⑥
    EXEC SQL PREPARE STMID FROM :str;                                    ⑦
    EXEC SQL ALLOCATE DESCRIPTOR 'INDESC' WITH MAX 100;                  ⑧
    EXEC SQL ALLOCATE DESCRIPTOR 'OUTDESC' WITH MAX 100;                 ⑨
    EXEC SQL DESCRIBE INPUT STMID USING SQL DESCRIPTOR 'INDESC';         ⑩
    EXEC SQL DESCRIBE OUTPUT STMID USING SQL DESCRIPTOR 'OUTDESC';       ⑪
```

(Continued)

```
/**************************/
/* Dynamic parameter specification */
/**************************/
EXEC SQL GET DESCRIPTOR 'INDESC' :varicount = COUNT;                   ⑫
for( i = 1; i <= varicount; i++ ) {
    EXEC SQL GET DESCRIPTOR 'INDESC' VALUE :i
                                 :vartype = TYPE,
                                 :varleng = LENGTH,
                                 :varolen = OCTET_LENGTH,
                                 :varprec = PRECISION,
                                 :varscal = SCALE,                     ⑬
                                 :varname = NAME,
                                 :varchar = CHARACTER_SET_NAME,
                                 :vardcod = DATETIME_INTERVAL_CODE,
                                 :vardpre = DATETIME_INTERVAL_PRECISION;
    /* CHAR type or NCHAR type */
    if( vartype == 1 || vartype == 12 ) {
        /* Character string type */
        if( strncmp( varchar, "BASIC", 5 ) == 0 ) {
            printf( " Specify the value for %s         \n", varname );
            scanf( "%s", icdata.sqlvar );
            icdata.sqllen = strlen( icdata.sqlvar );
            EXEC SQL SET DESCRIPTOR 'INDESC' VALUE :i
                                         TYPE = 1,
                                         LENGTH = 14,
                                         OCTET_LENGTH = 14,
                                         CHARACTER_SET_NAME = 'BASIC',
                                         DATA = :icdata;
        }
```

(Continued)

```
            /* National character string type */
            else {
                printf( " Specify the value for %s  \n", varname );
                scanf( "%s", incdata.sqlvar );
                incdata.sqllen = strlen( incdata.sqlvar );
                EXEC SQL SET DESCRIPTOR 'INDESC' VALUE :i
                                                 TYPE = 1,
                                                 LENGTH = 10,
                                                 OCTET_LENGTH = 20,
                                                 CHARACTER_SET_NAME = 'NCHAR',
                                                 DATA = :incdata;
            }
        }
        /* INTEGER type or SMALLINT type */
        else if( vartype == 4 || vartype == 5 ) {
            printf( " Specify the value for %s  \n", varname );
            scanf( "%ld", &iidata );
            EXEC SQL SET DESCRIPTOR 'INDESC' VALUE :i
                                             TYPE = 4,
                                             PRECISION = 31,
                                             SCALE = 0,
                                             INDICATOR = 0,
                                             DATA = :iidata;
        }
        /* NUMERIC type, DECIMAL type, REAL type, or DOUBLE PRECISION type */
        else if( vartype == 2 || vartype == 3 ||
                 vartype == 7 || vartype == 8 ) {
            printf( " Specify the value for %s  \n", varname );
            scanf( "%lf", &ifdata );
            EXEC SQL SET DESCRIPTOR 'INDESC' VALUE :i
                                             TYPE = 8,
                                             PRECISION = 52,
                                             SCALE = 0,
                                             DATA = :ifdata;
        }
```

⑭

(Continued)

```
        /* Datetime type or INTERVAL type  */
        else if( vartype == 9 || vartype == 10 ) {
            printf( "Specify the value for %s          \n", varname );
            scanf( "%s", iddata.sqlvar );
            iddata.sqllen = strlen( iddata.sqlvar );
            EXEC SQL SET DESCRIPTOR 'INDESC' VALUE :i
                                             TYPE = 12,
                                             LENGTH = 19,
                                             OCTET_LENGTH = 19,
                                             CHARACTER_SET_NAME = 'BASIC',
                                             DATA = :iddata;
        }
    }
    /**************/
    /*    Select list    */
    /**************/
    EXEC SQL GET DESCRIPTOR 'OUTDESC' :varocount = COUNT;                  ⑮
    for( i = 1; i <= varocount; i++ ) {
        EXEC SQL GET DESCRIPTOR 'OUTDESC' VALUE :i
                                         :vartype = TYPE,
                                         :varleng = LENGTH,
                                         :varolen = OCTET_LENGTH,
                                         :varprec = PRECISION,                ⑯
                                         :varscal = SCALE,
                                         :varchar = CHARACTER_SET_NAME,
                                         :vardcod = DATETIME_INTERVAL_CODE,
                                         :vardpre = DATETIME_INTERVAL_PRECISION;
        /* CHAR type or NCHAR type     */
        if( vartype == 1 || vartype == 12 ) {
            /* Character string type */
            if( strncmp( varchar, "BASIC", 5 ) == 0 ) {
                EXEC SQL SET DESCRIPTOR 'OUTDESC' VALUE :i
                                                  TYPE = 1,
                                                  LENGTH = 14,
                                                  OCTET_LENGTH = 14,
                                                  CHARACTER_SET_NAME = 'BASIC';
            }
```

(Continued)

```
            /* National character string type */
            else {
                EXEC SQL SET DESCRIPTOR 'OUTDESC' VALUE :i
                                                  TYPE = 1,
                                                  LENGTH = 10,
                                                  OCTET_LENGTH = 20,
                                                  CHARACTER_SET_NAME = 'NCHAR';
            }
        }
        /* INTEGER type or SMALLINT type */
        else if( vartype == 4 || vartype == 5 ) {
            EXEC SQL SET DESCRIPTOR 'OUTDESC' VALUE :i                  ⑰
                                              TYPE = 4,
                                              PRECISION = 31,
                                              SCALE = 0;
        }
        /* NUMERIC type, DECIMAL type, REAL type, or DOUBLE PRECISION type */
        else if( vartype == 2 || vartype == 3 ||
                 vartype == 7 || vartype == 8 ) {
            EXEC SQL SET DESCRIPTOR 'OUTDESC' VALUE :i
                                              TYPE = 8,
                                              PRECISION = 52,
                                              SCALE = 0;
        }
        /* Datetime type or INTERVAL type */
        else if( vartype == 9 || vartype == 10 ) {
            EXEC SQL SET DESCRIPTOR 'OUTDESC' VALUE :i
                                              TYPE = 12,
                                              LENGTH = 19,
                                              OCTET_LENGTH = 19,
                                              CHARACTER_SET_NAME = 'BASIC';
        }
    }
```

(Continued)

```
EXEC SQL OPEN CUR1 USING SQL DESCRIPTOR 'INDESC' ;                    ⑱
EXEC SQL WHENEVER NOT FOUND GOTO :NOTFND;                             ⑲
for(;;) {
    EXEC SQL FETCH CUR1 INTO SQL DESCRIPTOR 'OUTDESC' ;               ⑳
    /*******************/
    /*    Execution result    */
    /*******************/
    for( i = 1; i <= varocount; i++ ) {
        EXEC SQL GET DESCRIPTOR 'OUTDESC' VALUE :i
                                          :vartype = TYPE,
                                          :varchar = CHARACTER_SET_NAME;
        /* CHAR type or NCHAR type  */
        if( vartype == 1 ) {
            /* Character string type */
            if( strncmp( varchar, "BASIC", 5 ) == 0 ) {
                EXEC SQL GET DESCRIPTOR 'OUTDESC' VALUE :i
                                                  :varname = NAME,
                                                  :ocdata = DATA;
                printf( "%s -> %s   ", varname, ocdata );
            }
            /* National character string type */
            else {
                EXEC SQL GET DESCRIPTOR 'OUTDESC' VALUE :i
                                                 :varname = NAME,
                                                 :oncdata = DATA;
                printf( "%s -> %s   ", varname, oncdata );
            }
        }                                                                ㉑
        /* Exact numeric value type  */
        else if( vartype == 4 ) {
            EXEC SQL GET DESCRIPTOR 'OUTDESC' VALUE :i
                                              :varname = NAME,
                                              :oidata = DATA;
            printf( "%s -> %d   ", varname, oidata );
        }
```

(Continued)

```
            /* Approximate numeric value type     */
            else if( vartype == 8 ) {
                EXEC SQL GET DESCRIPTOR 'OUTDESC' VALUE :i
                                                  :varname = NAME,
                                                  :ofdata = DATA;
                printf( "%s -> %lf   ", varname, ofdata );
            }
            /* Datetime type or INTERVAL type  */
            else if( vartype == 12 ) {
                EXEC SQL GET DESCRIPTOR 'OUTDESC' VALUE :i
                                                  :varname = NAME,
                                                  :oddata = DATA;
                printf( "%s -> %s   ", varname, oddata.sqlvar );
            }
        }
        printf( "\n" );
    }
NOTFND:
    EXEC SQL CLOSE CUR1;                                                        ㉒
    EXEC SQL DEALLOCATE DESCRIPTOR 'INDESC';                                    ㉓
    EXEC SQL DEALLOCATE DESCRIPTOR 'OUTDESC';                                   ㉔
    EXEC SQL DEALLOCATE PREPARE STMID;                                          ㉕
    EXEC SQL COMMIT WORK;                                                       ㉖
    EXEC SQL DISCONNECT DEFAULT;                                                ㉗
    return;
ERR:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    printf( "SQLERROR SQLSTATE = %s SQLMSG = %s\n", SQLSTATE, SQLMSG );
    EXEC SQL ROLLBACK WORK;
    EXEC SQL DISCONNECT DEFAULT;
    return;
}
```

(Continued)

① The host variable of include file varfile.h is expanded. The SQL statement variable is the area in which the SQL statement is stored. A variable-length character string is expanded into the following structure when the program is compiled:

```
        struct {
            short  sqllen;        /* Length area variable           */
            char   sqlvar[100];   /* Character string area variable */
        }str;
```

② Initializes host variables.
③ Opens a connection.
④ If an error other than "no data" occurs during execution of the SQL statement, the program branches to ERR, outputs error information, and terminates.
⑤ Reads the dynamic SELECT statement entered from the terminal and stores it in str.
⑥ Declares cursor CUR1. Specifies SQL statement identifier STMID as the cursor specification.
⑦ Prepares the SQL statement in str for execution.
⑧ Allocates SQL descriptor area INDESC for the dynamic parameter specification.
⑨ Allocates SQL descriptor area OUTDESC for the target specification.
⑩ Fetches dynamic parameter specification information for the input dynamic SELECT statement to SQL descriptor area INDESC.
⑪ Fetches select list information for the input dynamic SELECT statement to SQL descriptor area OUTDESC.
⑫ Gets the number of dynamic parameter specifications (COUNT value) set in INDESC.
⑬ Gets the information for each dynamic parameter specification set in INDESC (TYPE, LENGTH, OCTET_LENGTH, PRECISION, SCALE, CHARACTER_SET_NAME, DATETIME_INTERVAL_CODE, DATETIME_INTERVAL_PRECISION values).
⑭ Modifies the data type of the dynamic parameter specification set in INDESC, and sets the value of the dynamic parameter specification read from the terminal.

- Changes CHARACTER(n) and CHARACTER VARYING(n) to CHARACTER(14). (n indicates the length of the character string.)
- Changes NATIONAL CHARACTER(n) and NATIONAL CHARACTER VARYING(n) to NATIONAL CHARACTER(10). (n indicates the length of the character string.)
- Changes INTEGER and SMALLINT to INTEGER.
- Changes NUMERIC(p, q), DECIMAL(p, q), REAL, and DOUBLE PRECISION to DOUBLE PRECISION. (p indicates the precision and q indicates the scale.)
- Changes DATE, TIME, TIMESTAMP, and INTERVAL to CHARACTER VARYING(19). (n indicates the length of the character string.)

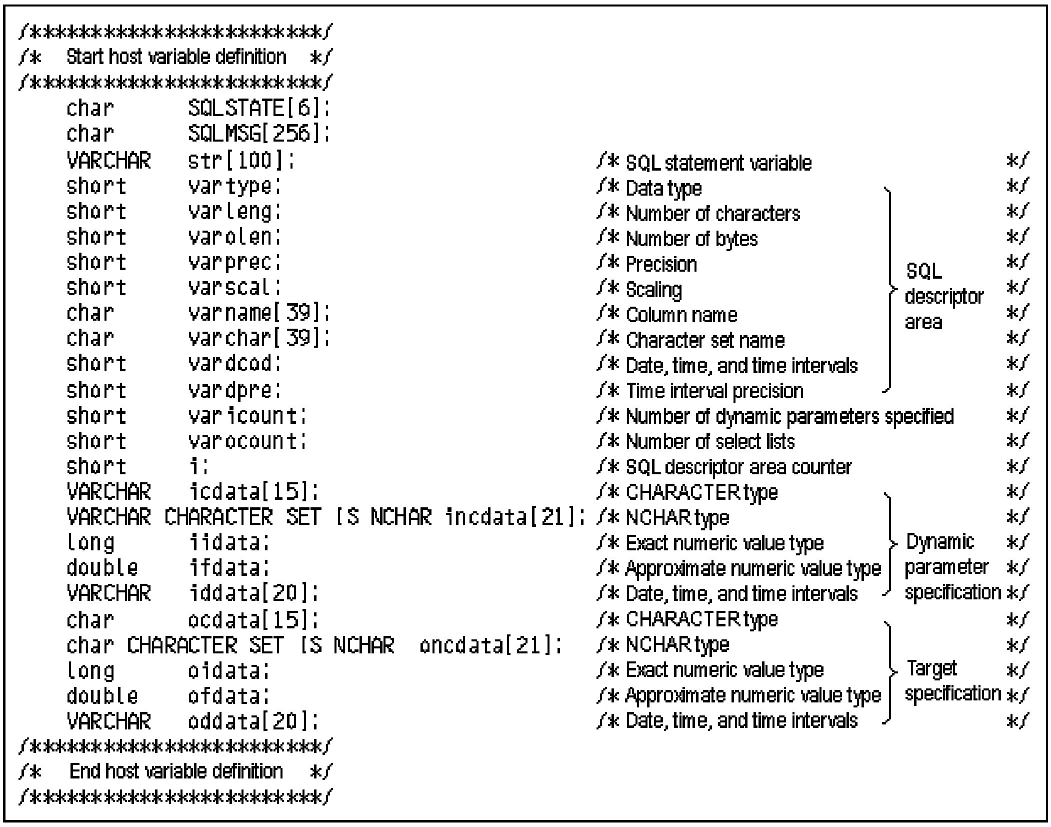
⑮ Gets the number of select lists (COUNT value) set in OUTDESC.
⑯ Gets the information for each select list (TYPE, LENGTH, OCTET_LENGTH, PRECISION, SCALE, CHARACTER_SET_NAME, DATETIME_INTERVAL_CODE, DATETIME_INTERVAL_PRECISION values) set in OUTDESC.
⑰ Sets the data type of the target specification from which the execution result is fetched to OUTDESC.

- Changes CHARACTER(n) and CHARACTER VARYING(n) to CHARACTER(14). (n indicates the length of the character string.)
- Changes NATIONAL CHARACTER(n) and NATIONAL CHARACTER VARYING(n) to NATIONAL CHARACTER(10). (n indicates the length of the character string.)
- Changes INTEGER and SMALLINT to INTEGER.
- Changes NUMERIC(p, q), DECIMAL(p, q), REAL, and DOUBLE PRECISION to DOUBLE PRECISION. (p indicates the precision and q indicates the scale.)
- Changes DATE, TIME, TIMESTAMP, and INTERVAL to CHARACTER VARYING(19). (n indicates the length of the character string.)

⑱ Opens cursor CUR1 using the value obtained in ⑭.
⑲ If search result data is not found, the program branches to NOTFND and terminates.
⑳ Positions cursor CUR1 and reads data. The execution result value is set at the address specified in OUTDESC DATA.
㉑ Gets the NAME and DATA values from SQL descriptor area OUTDESC. Each value is stored in the host variable that matches the data type of the target specification.
㉒ Closes cursor CUR1.
㉓ Deallocates SQL descriptor area INDESC.
㉔ Deallocates SQL descriptor area OUTDESC.
㉕ Deallocates the prepared statement corresponding to SQL statement identifier STMID.
㉖ Terminates the current transaction.
㉗ Closes the connection

(Continued)

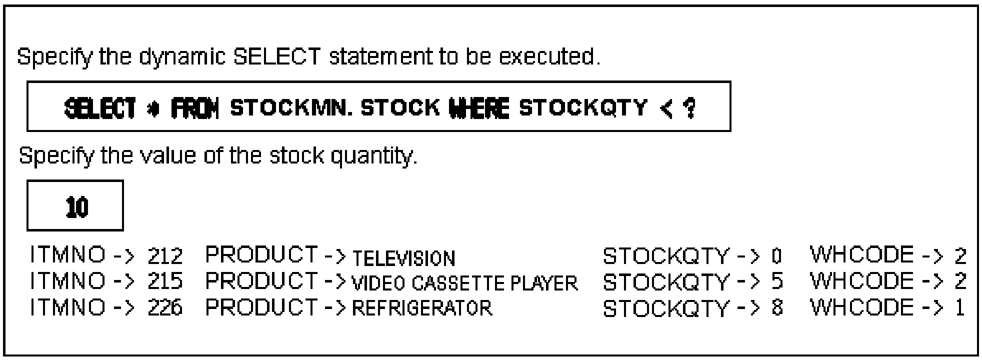
Include file "varfile.h"



```
/*************************/
/*  Start host variable definition  */
/*************************/
    char      SQLSTATE[6];
    char      SQLMSG[256];
    VARCHAR   str[100];                    /* SQL statement variable                       */
    short     vartype;                     /* Data type                                    */
    short     varleng;                     /* Number of characters                         */
    short     varolen;                     /* Number of bytes                              */
    short     varprec;                     /* Precision                       SQL          */
    short     varscal;                     /* Scaling                         descriptor   */
    char      varname[39];                 /* Column name                     area         */
    char      varchar[39];                 /* Character set name                           */
    short     vardcod;                     /* Date, time, and time intervals               */
    short     vardpre;                     /* Time interval precision                      */
    short     varicount;                   /* Number of dynamic parameters specified       */
    short     varocount;                   /* Number of select lists                       */
    short     i;                           /* SQL descriptor area counter                  */
    VARCHAR   icdata[15];                  /* CHARACTER type                               */
    VARCHAR CHARACTER SET IS NCHAR incdata[21]; /* NCHAR type                              */
    long      iidata;                      /* Exact numeric value type        Dynamic      */
    double    ifdata;                      /* Approximate numeric value type  parameter    */
    VARCHAR   iddata[20];                  /* Date, time, and time intervals  specification */
    char      ocdata[15];                  /* CHARACTER type                               */
    char CHARACTER SET IS NCHAR  oncdata[21]; /* NCHAR type                                */
    long      oidata;                      /* Exact numeric value type        Target       */
    double    ofdata;                      /* Approximate numeric value type  specification */
    VARCHAR   oddata[20];                  /* Date, time, and time intervals               */
/*************************/
/*  End host variable definition  */
/*************************/
```

(Continued)

Remarks: A sample image output when the execution of this application program terminates normally is shown below. Sans serif character strings indicate input.



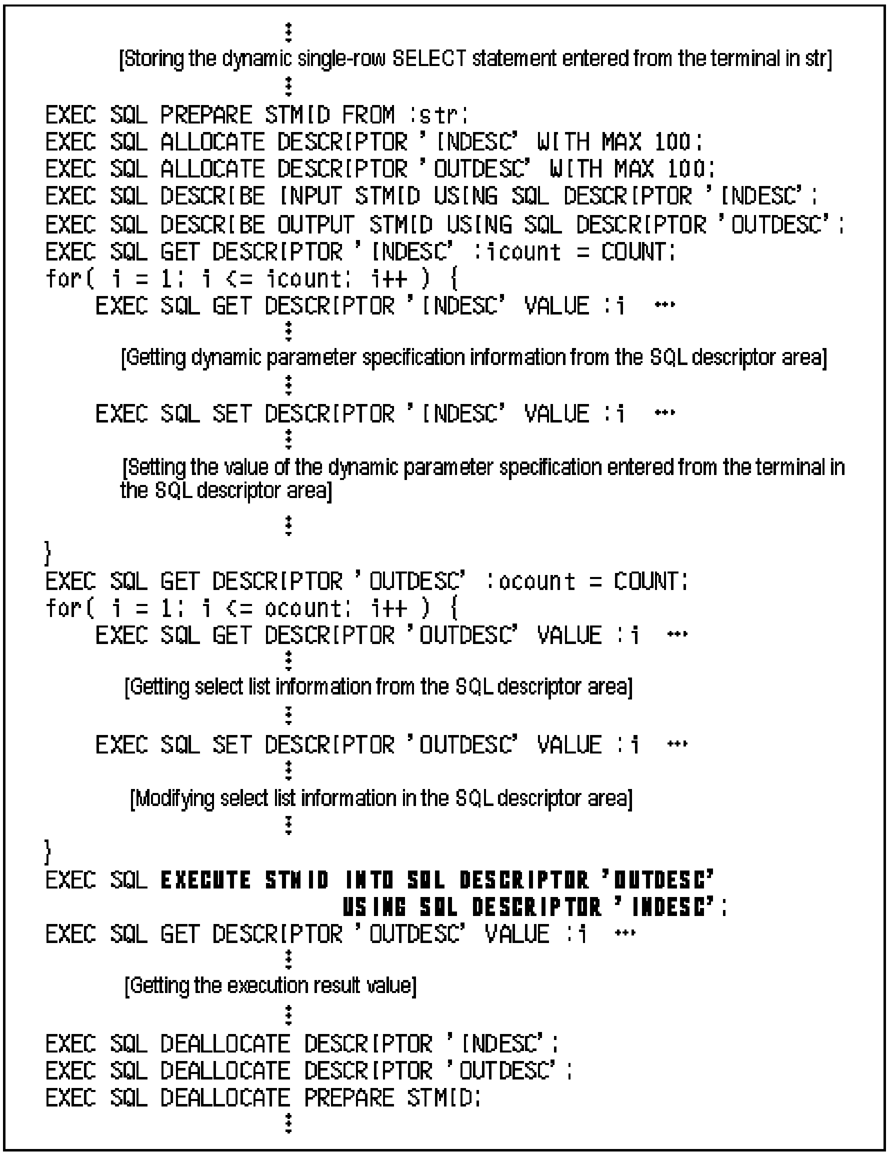
```
Specify the dynamic SELECT statement to be executed.
 SELECT * FROM STOCKMN. STOCK WHERE STOCKQTY < ?
Specify the value of the stock quantity.
 10
ITMNO -> 212  PRODUCT -> TELEVISION               STOCKQTY -> 0  WHCODE -> 2
ITMNO -> 215  PRODUCT -> VIDEO CASSETTE PLAYER    STOCKQTY -> 5  WHCODE -> 2
ITMNO -> 226  PRODUCT -> REFRIGERATOR             STOCKQTY -> 8  WHCODE -> 1
```

7.3.2.2 Preparing and executing dynamic single-row SELECT statements (for SQL descriptor area)

To fetch single-row data, prepare and execute a dynamic single-row SELECT statement. To execute a dynamic single-row SELECT statement, use the EXECUTE statement with the USING clause specified. An example of using the SQL descriptor area to specify a dynamic single-row SELECT statement with the dynamic parameter specification is shown below. For information about SQL statements used to manipulate the SQL descriptor area, see "Figure: Procedure for setting dynamic parameter specification values for prepared statements."

Example:

> This is an example of a dynamic single-row SELECT statement entered from a terminal. Data is searched using the value of the dynamic parameter specification entered from the terminal as the search condition.



```
                      ⋮
        [Storing the dynamic single-row SELECT statement entered from the terminal in str]
                      ⋮
EXEC SQL PREPARE STMID FROM :str;
EXEC SQL ALLOCATE DESCRIPTOR 'INDESC' WITH MAX 100;
EXEC SQL ALLOCATE DESCRIPTOR 'OUTDESC' WITH MAX 100;
EXEC SQL DESCRIBE INPUT STMID USING SQL DESCRIPTOR 'INDESC';
EXEC SQL DESCRIBE OUTPUT STMID USING SQL DESCRIPTOR 'OUTDESC';
EXEC SQL GET DESCRIPTOR 'INDESC' :icount = COUNT;
for( i = 1; i <= icount; i++ ) {
    EXEC SQL GET DESCRIPTOR 'INDESC' VALUE :i  ...
                      ⋮
        [Getting dynamic parameter specification information from the SQL descriptor area]
                      ⋮
    EXEC SQL SET DESCRIPTOR 'INDESC' VALUE :i  ...
                      ⋮
        [Setting the value of the dynamic parameter specification entered from the terminal in
        the SQL descriptor area]
                      ⋮
}
EXEC SQL GET DESCRIPTOR 'OUTDESC' :ocount = COUNT;
for( i = 1; i <= ocount; i++ ) {
    EXEC SQL GET DESCRIPTOR 'OUTDESC' VALUE :i  ...
                      ⋮
        [Getting select list information from the SQL descriptor area]
                      ⋮
    EXEC SQL SET DESCRIPTOR 'OUTDESC' VALUE :i  ...
                      ⋮
        [Modifying select list information in the SQL descriptor area]
                      ⋮
}
EXEC SQL EXECUTE STMID INTO SQL DESCRIPTOR 'OUTDESC'
                       USING SQL DESCRIPTOR 'INDESC';
EXEC SQL GET DESCRIPTOR 'OUTDESC' VALUE :i  ...
                      ⋮
        [Getting the execution result value]
                      ⋮
EXEC SQL DEALLOCATE DESCRIPTOR 'INDESC';
EXEC SQL DEALLOCATE DESCRIPTOR 'OUTDESC';
EXEC SQL DEALLOCATE PREPARE STMID;
                      ⋮
```

7.3.3 Executing other prepared statements

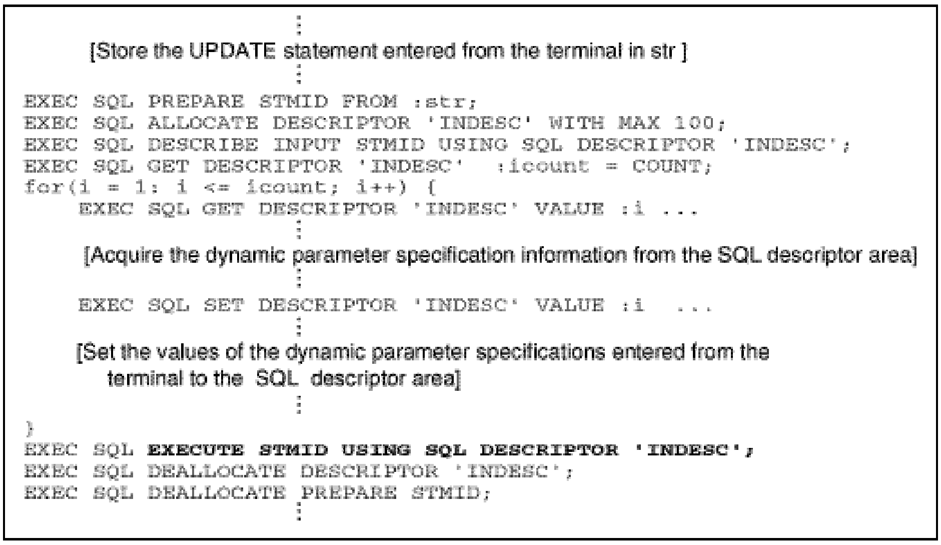
If a dynamic parameter specification is specified to perform data manipulations such as inserting or updating data, use an EXECUTE statement in which a USING clause is specified. The prepared statements that can be executed by an EXECUTE statement are preparable statements other than dynamic SELECT statements. For more details on statements that can be used for dynamic execution, see 7.1 "Overview of Dynamic SQL".

If the SQL statement variable contains an UPDATE statement (positioned) or a DELETE statement (positioned), the corresponding cursor must be opened beforehand. Also, the specified cursor must be a dynamic cursor.

An example of specifying an UPDATE statement (searched) with a dynamic parameter follows. Refer to the example in Figure: Example of an application program that uses the SQLDA structure (with dynamic parameter specification) for details on SQL statements for manipulating the SQL descriptor area.

Example:

This is an example of an UPDATE statement (searched) entered from a terminal. Data for the values of the dynamic parameter specifications entered from the terminal is updated as a set clause.



```
                         ⋮
      [Store the UPDATE statement entered from the terminal in str ]
                         ⋮
EXEC SQL PREPARE STMID FROM :str;
EXEC SQL ALLOCATE DESCRIPTOR 'INDESC' WITH MAX 100;
EXEC SQL DESCRIBE INPUT STMID USING SQL DESCRIPTOR 'INDESC';
EXEC SQL GET DESCRIPTOR 'INDESC'  :icount = COUNT;
for(i = 1: i <= icount; i++) {
    EXEC SQL GET DESCRIPTOR 'INDESC' VALUE :i ...
                         ⋮
     [Acquire the dynamic parameter specification information from the SQL descriptor area]
                         ⋮
    EXEC SQL SET DESCRIPTOR 'INDESC' VALUE :i  ...
                         ⋮
    [Set the values of the dynamic parameter specifications entered from the
       terminal to the  SQL  descriptor area]
                         ⋮
}
EXEC SQL EXECUTE STMID USING SQL DESCRIPTOR 'INDESC';
EXEC SQL DEALLOCATE DESCRIPTOR 'INDESC';
EXEC SQL DEALLOCATE PREPARE STMID;
                         ⋮
```

7.3.4 Executing prepared statements for which variable attributes are known

When an application program is created, the attributes and the number of dynamic parameter specifications for prepared statements, or the attributes and the number of select column lists to be fetched may be known. If so, the SQL descriptor area need not be used and the values can be set or the execution results fetched by specifying USING arguments.

To set the values of dynamic parameter specifications for a prepared statement by specifying USING arguments, the values to be set in the argument must be assigned. The number of dynamic parameter specifications in the prepared statement and the number of arguments must match. The data type of the dynamic parameter specification and that of the arguments must be comparable. To specify a null value, use an indicator variable.

To fetch the execution results for the prepared statement by specifying USING arguments, specify an argument that will receive the execution results. The number of select column lists in the prepared statement and the number of arguments must match. The data type of the select column lists and that of the arguments must be comparable. For information about comparable data types, see Table "Comparable data types".

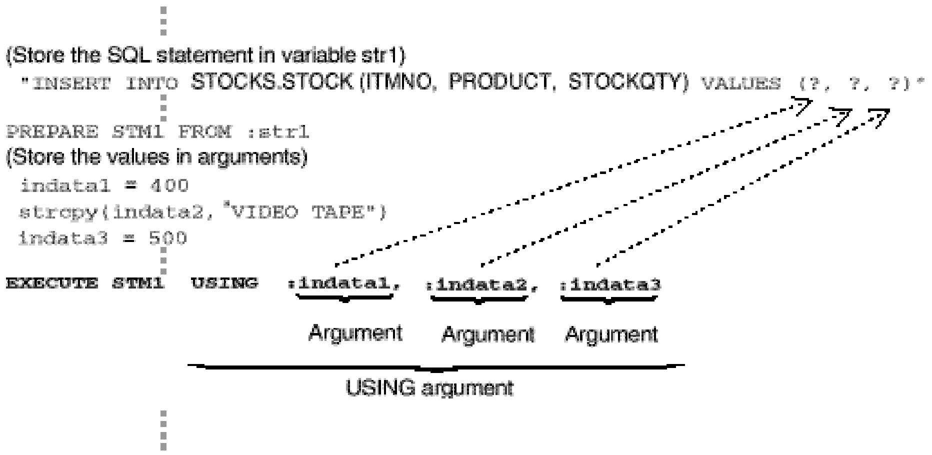
Various specification examples follow.

■Using the EXECUTE statement to set the values of dynamic parameter specifications

Specify USING arguments in the EXECUTE statement to set the values of dynamic parameter specifications.

Example 1:

In this example, a row for which ITMNO is "400", PRODUCT is "VIDEO TAPE", and STOCKQTY is "500" is added to the STOCK table. The data types of ITMNO, PRODUCT, and STOCKQTY and those of the three arguments in which the values are to be set must be comparable.



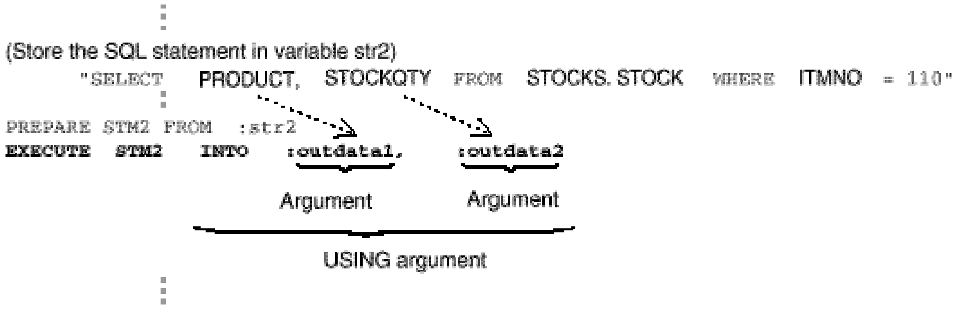
```
(Store the SQL statement in variable str1)
 "INSERT INTO STOCKS.STOCK (ITMNO, PRODUCT, STOCKQTY) VALUES (?, ?, ?)"

PREPARE STM1 FROM :str1
(Store the values in arguments)
 indata1 = 400
 strcpy(indata2, "VIDEO TAPE")
 indata3 = 500

EXECUTE STM1  USING  :indata1,  :indata2,  :indata3
```

■Using the EXECUTE statement to fetch execution results

Specify USING arguments in the EXECUTE statement to fetch execution results.

Example 2:

In this example, the values of PRODUCT and STOCKQTY for the product with ITMNO "110" are fetched from the STOCK table. The data types for PRODUCT and STOCKQTY and those for the two arguments in which the results are fetched must be comparable.



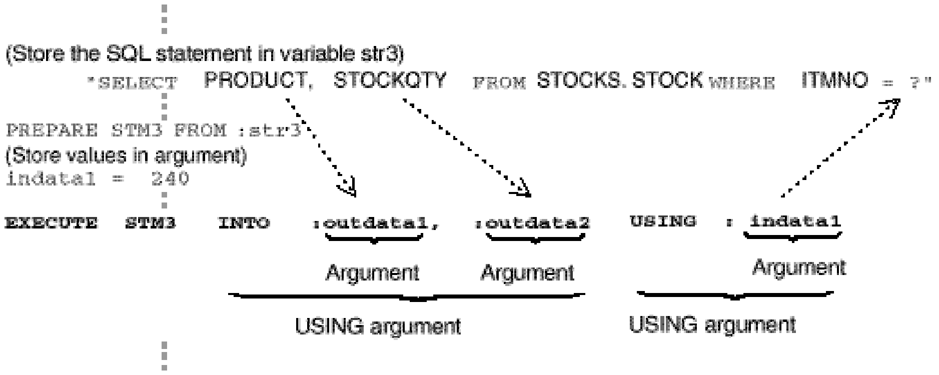
```
(Store the SQL statement in variable str2)
     "SELECT  PRODUCT, STOCKQTY  FROM  STOCKS. STOCK  WHERE  ITMNO = 110"

PREPARE STM2 FROM  :str2
EXECUTE  STM2  INTO  :outdata1,   :outdata2
```

■Using the EXECUTE statement to set the values of dynamic parameter specifications and to fetch execution results

Specify USING arguments in the EXECUTE statement to set the values of dynamic parameter specifications and to fetch execution results.

Example 3:

In this example, the values of PRODUCT and STOCKQTY for the product with ITMNO "240" are fetched from the STOCK table. The data types of ITMNO and the argument in which the value is set, and the data types of PRODUCT and STOCKQTY and the two arguments in which the results are fetched must be comparable.



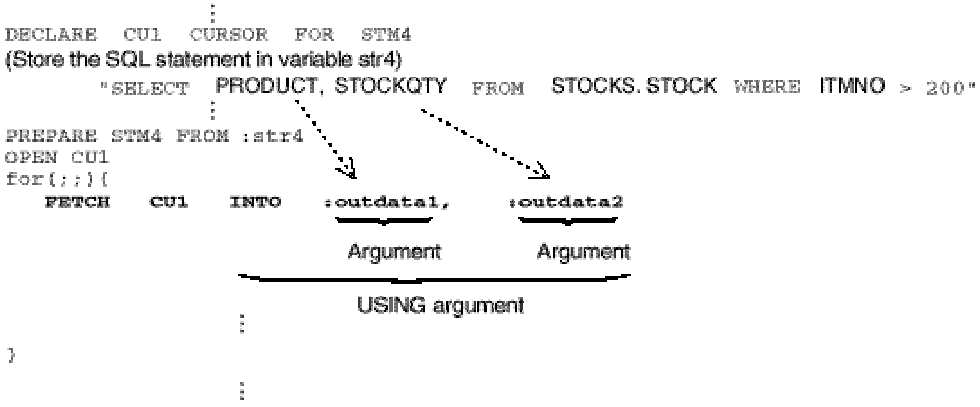
```
      ⋮
(Store the SQL statement in variable str3)
      "SELECT  PRODUCT, STOCKQTY  FROM STOCKS.STOCK WHERE  ITMNO = ?"
      ⋮
PREPARE STM3 FROM :str3
(Store values in argument)
indata1 =  240
      ⋮
EXECUTE  STM3   INTO   :outdata1,   :outdata2   USING  : indata1
      ⋮
```

■Using the dynamic FETCH statement to fetch execution results

Specify USING arguments in the dynamic FETCH statement to fetch execution results.

Example 4:

In this example, values of PRODUCT and STOCKQTY with ITMNO greater than "200" are fetched from the STOCK table. The data types of ITMNO and STOCKQTY and those of the two arguments in which the results are fetched must be comparable.



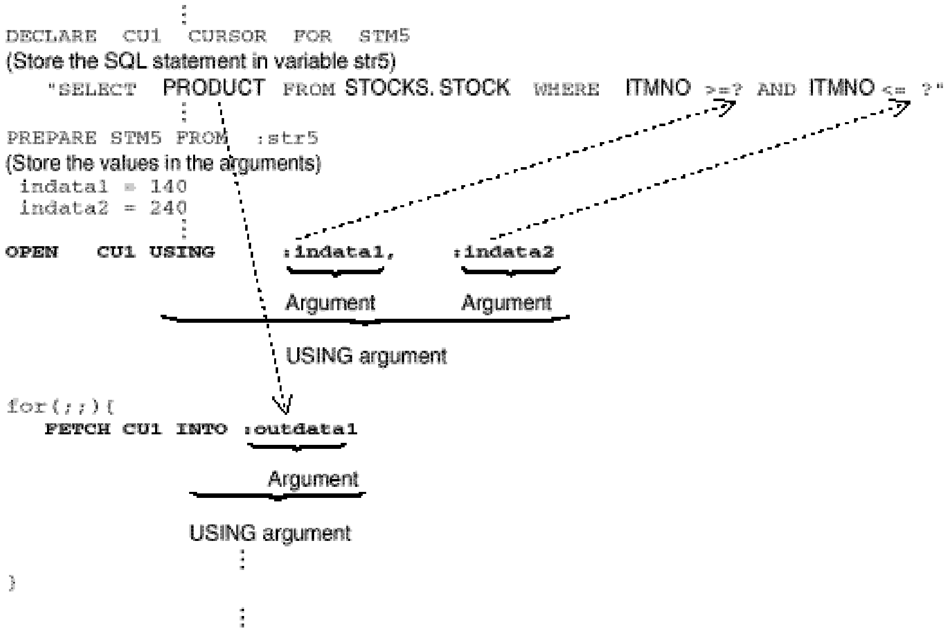
```
                 ⋮
DECLARE  CU1  CURSOR  FOR  STM4
(Store the SQL statement in variable str4)
      "SELECT  PRODUCT, STOCKQTY  FROM  STOCKS.STOCK  WHERE  ITMNO > 200"
                 ⋮
PREPARE STM4 FROM :str4
OPEN CU1
for(;;){
   FETCH   CU1   INTO   :outdata1,    :outdata2
                   ⋮
}
                   ⋮
```

■Using the dynamic OPEN statement to set the values of dynamic parameter specifications and the dynamic FETCH statement to fetch execution results

Specify USING arguments in the dynamic OPEN statement and the dynamic FETCH statement to set the values of dynamic parameter specifications and fetch execution results.

Example 5:

In this example, values for PRODUCT with ITMNO between "140" and "240" inclusive are fetched from the STOCK table. The data types of ITMNO, the two arguments in which the values are set and the argument to which the result is fetched must be comparable.



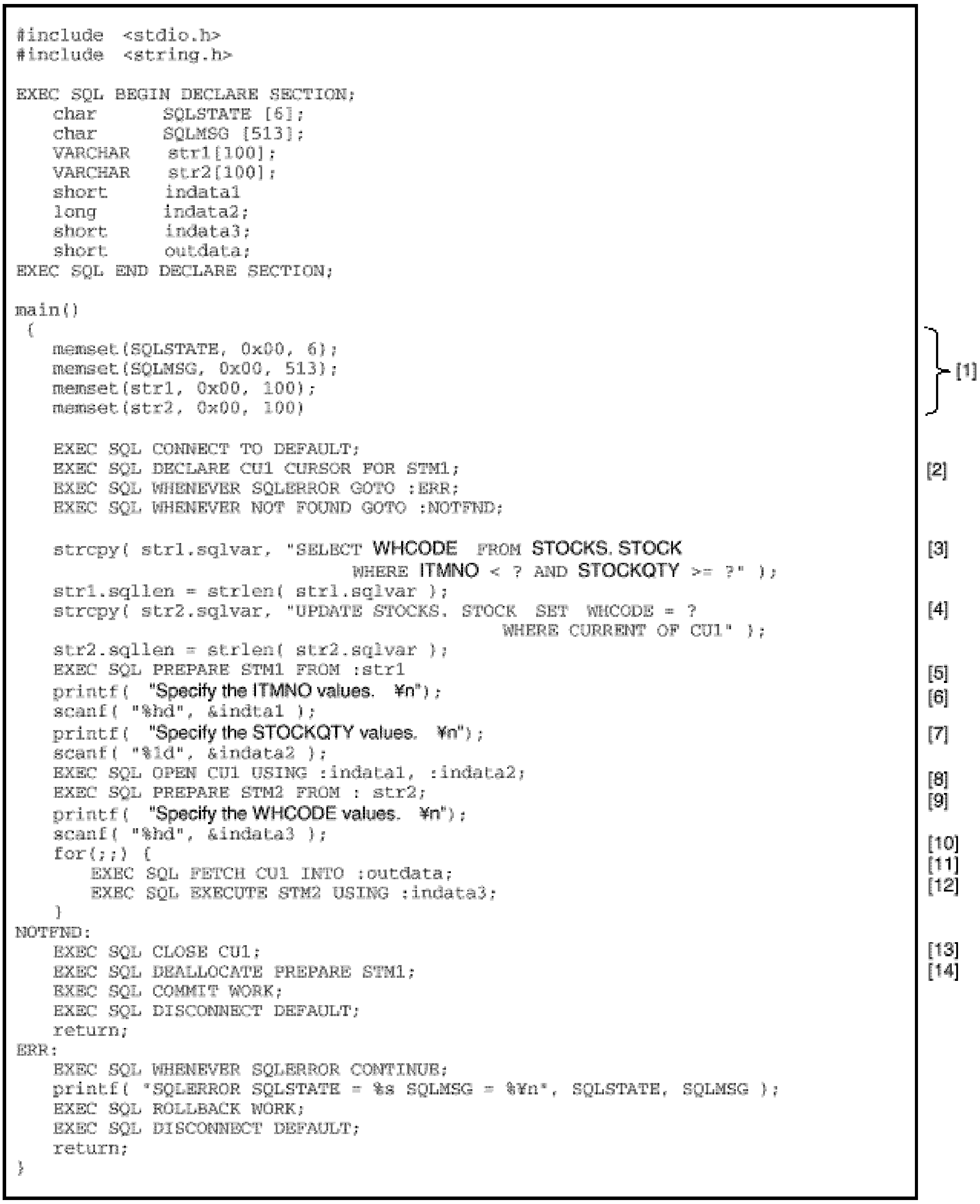
```
      :
DECLARE  CU1  CURSOR  FOR  STM5
(Store the SQL statement in variable str5)
   "SELECT  PRODUCT  FROM STOCKS.STOCK  WHERE  ITMNO >=? AND ITMNO <= ?"
      :
PREPARE STM5 FROM  :str5
(Store the values in the arguments)
 indata1 = 140
 indata2 = 240
      :
OPEN   CU1 USING     :indata1,    :indata2

for(;;){
   FETCH CU1 INTO :outdata1
          :
}
          :
```

Figure: Example of an application program that uses the USING argument is an example of an application program that uses the USING argument.

Example 6:

> This is an example of a dynamic SELECT statement that has two dynamic parameter specifications. Values for ITMNO and STOCKQTY entered from a terminal are used as search conditions to fetch data. The rows that satisfy the search conditions are updated to the value of WHCODE entered from the terminal.

[Figure: Example of an application program that uses the USING argument]



```
#include  <stdio.h>
#include  <string.h>

EXEC SQL BEGIN DECLARE SECTION;
    char      SQLSTATE [6];
    char      SQLMSG [513];
    VARCHAR    str1[100];
    VARCHAR    str2[100];
    short     indata1
    long      indata2;
    short     indata3;
    short     outdata;
EXEC SQL END DECLARE SECTION;

main()
 {
    memset(SQLSTATE, 0x00, 6);                                                  ┐
    memset(SQLMSG, 0x00, 513);                                                  ├ [1]
    memset(str1, 0x00, 100);                                                    │
    memset(str2, 0x00, 100)                                                     ┘

    EXEC SQL CONNECT TO DEFAULT;
    EXEC SQL DECLARE CU1 CURSOR FOR STM1;                                        [2]
    EXEC SQL WHENEVER SQLERROR GOTO :ERR;
    EXEC SQL WHENEVER NOT FOUND GOTO :NOTFND;

    strcpy( str1.sqlvar, "SELECT WHCODE  FROM STOCKS. STOCK                      [3]
                               WHERE ITMNO < ? AND STOCKQTY >= ?" );
    str1.sqllen = strlen( str1.sqlvar );
    strcpy( str2.sqlvar, "UPDATE STOCKS. STOCK  SET  WHCODE = ?                  [4]
                                           WHERE CURRENT OF CU1" );
    str2.sqllen = strlen( str2.sqlvar );
    EXEC SQL PREPARE STM1 FROM :str1                                             [5]
    printf(  "Specify the ITMNO values.   ¥n");                                  [6]
    scanf( "%hd", &indta1 );
    printf(  "Specify the STOCKQTY values.   ¥n");                               [7]
    scanf( "%ld", &indata2 );
    EXEC SQL OPEN CU1 USING :indata1, :indata2;                                  [8]
    EXEC SQL PREPARE STM2 FROM : str2;                                           [9]
    printf(  "Specify the WHCODE values.   ¥n");
    scanf( "%hd", &indata3 );                                                    [10]
    for(;;) {                                                                    [11]
        EXEC SQL FETCH CU1 INTO :outdata;                                        [12]
        EXEC SQL EXECUTE STM2 USING :indata3;
    }
NOTFND:
    EXEC SQL CLOSE CU1;                                                          [13]
    EXEC SQL DEALLOCATE PREPARE STM1;                                            [14]
    EXEC SQL COMMIT WORK;
    EXEC SQL DISCONNECT DEFAULT;
    return;
ERR:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    printf( "SQLERROR SQLSTATE = %s SQLMSG = %¥n", SQLSTATE, SQLMSG );
    EXEC SQL ROLLBACK WORK;
    EXEC SQL DISCONNECT DEFAULT;
    return;
}
```

[1] The host variables are initialized.

[2] Cursor CUR1 is declared. The SQL statement identifier STMID is specified as the cursor specification.

[3] The SQL statement (dynamic SELECT statement) is stored in SQL statement variable str1.

[4] The SQL statement (UPDATE statement (positioned)) is stored in SQL statement variable str2.

[5] The SQL statement in str is prepared for execution. STM1 is the SQL statement identifier.

[6] The values of dynamic parameter specifications for ITMNO are entered from a terminal and stored in host variable indata1.

[7] The values of dynamic parameter specifications for STOCKQTY are entered from a terminal and stored in host variable indata2.

[8] Use the values in [6] and [7] and open cursor CU1.

[9] The SQL statement in str2 is prepared for execution. STM2 is the SQL statement identifier.

[10] The values of dynamic parameter specifications for WHCODE are entered from a terminal and stored in host variable indata3.

[11] Cursor CU1 is positioned and the data is read. The execution result values are set in host variable outdata.

[12] The row where the cursor was positioned by the dynamic FETCH statement is updated to the value in [10].

[13] Cursor CU1 is closed.

[14] The prepared statement (dynamic SELECT statement) corresponding to SQL statement identifier STM1 is released. The prepared statement (UPDATE statement (positioned)) that is referencing cursor CU1 corresponding to STM1 is released at the same time.

Note: If “200” is specified for ITMNO, “100” for STOCKQTY, and “3” for WHCODE, the result is as follows:

STOCK table (before updating)

ITMNO	PRODUCT	STOCKQTY	WHCODE
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
140	CASSETTE DECK	120	2
212	TELEVISION	0	2
215	VIDEO CASSETTE PLAYER	5	2
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETTE TAPE	2500	2

⇨

STOCK table (after updating)

ITMNO	PRODUCT	STOCKQTY	WHCODE
110	TELEVISION	85	2
111	TELEVISION	90	2
123	REFRIGERATOR	60	1
124	REFRIGERATOR	75	1
140	CASSETTE DECK	120	3
212	TELEVISION	0	2
215	VIDEO CASSETTE PLAYER	0	2
226	REFRIGERATOR	8	1
227	REFRIGERATOR	15	1
240	CASSETTE DECK	25	2
243	CASSETTE DECK	14	2
351	CASSETTE TAPE	2500	2

7.4 Immediately Executing SQL Statements

If an SQL statement is to be executed without being prepared, use the EXECUTE IMMEDIATE statement. The SQL statements that can be executed using an EXECUTE IMMEDIATE statement are preparable statements other than the dynamic SELECT statement and the single row SELECT statement. For details on preparable statements, see 7.1

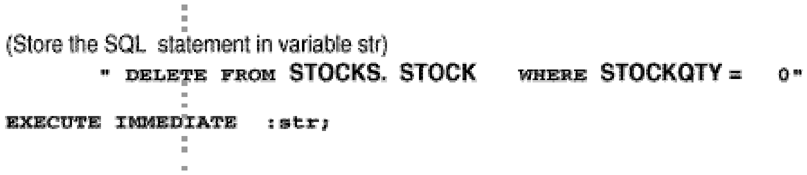
"Overview of Dynamic SQL". When the preparable statement is the INSERT statement, UPDATE statement (searched), DELETE statement (searched), UPDATE statement (positioned), or DELETE statement (positioned), the dynamic parameter specification cannot be specified.

If an SQL statement variable contains an UPDATE statement (positioned) or a DELETE statement (positioned), the corresponding cursor must be opened beforehand. Also, the specified cursor must be a dynamic cursor.

The following is an example of specifying an EXECUTE IMMEDIATE statement.

Example:

The rows for which STOCKQTY is "0" are deleted from the STOCK table.



```
                   ⋮
(Store the SQL  statement in variable str)
      " DELETE FROM STOCKS. STOCK    WHERE STOCKQTY =   0"
                   ⋮
EXECUTE IMMEDIATE  :str;
                   ⋮
```

7.5 Changing Database Names and Omitted Schema Names

This section explains how to change database names and omitted schema names in prepared statements to be executed by dynamic SQL in an application program.

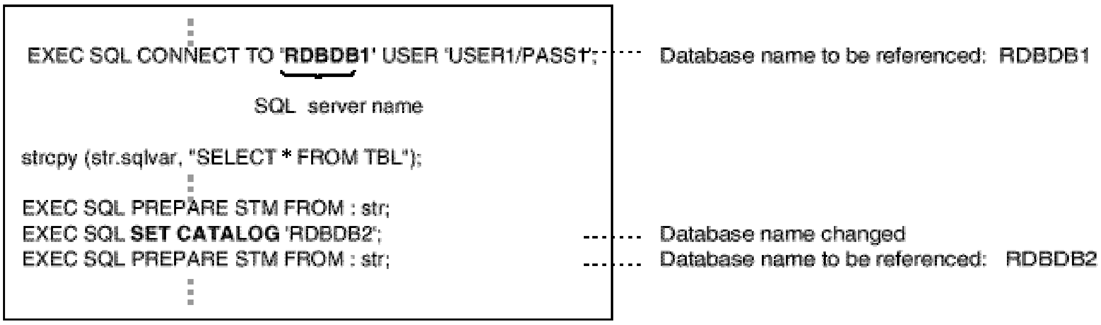
7.5.1 Changing database names

The database name for a prepared statement executed by dynamic SQL is the database name furnished by the connection. The furnished database name is the name of the SQL server name specified in the CONNECT statement. This database name can be changed in the application program. The database name specified in the SET CATALOG statement is for the prepared statements of PREPARE and EXECUTE IMMEDIATE statements. The changed database name is valid until the session terminates.

The following is an example of specifying the SET CATALOG statement:

Example:

In this example, the name of the data base is changed to "RDBDB2".



```
                 ⋮
EXEC SQL CONNECT TO 'RDBDB1' USER 'USER1/PASS1';
strcpy (str.sqlvar, "SELECT * FROM TBL");
                 ⋮
EXEC SQL PREPARE STM FROM : str;
EXEC SQL SET CATALOG 'RDBDB2';
EXEC SQL PREPARE STM FROM : str;
                 ⋮
```

SQL server name
Database name to be referenced: RDBDB1
Database name changed
Database name to be referenced: RDBDB2

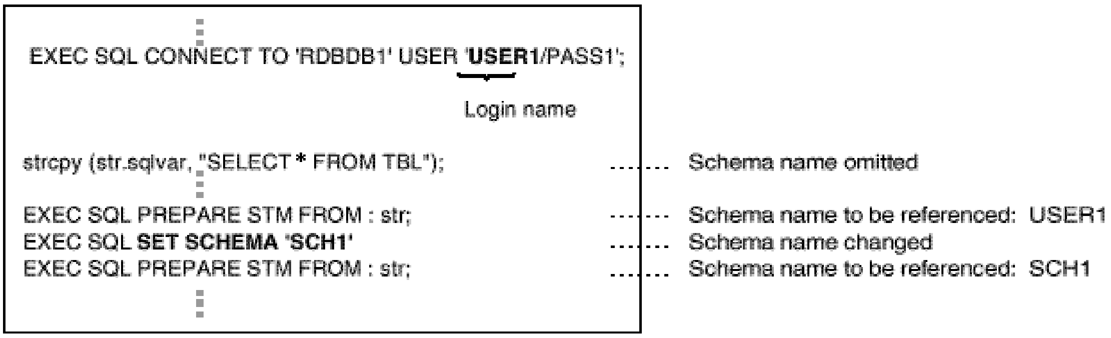
7.5.2 Changing schema names

For more information on omitting schema name qualifiers when a table declaration is specified, see 6.10 "Omitting Schema Names". A schema name based on a table declaration cannot be omitted from the table name of a prepared statement executed by dynamic SQL. If the schema name is omitted from the table name of a prepared statement, the login name specified in the CONNECT statement becomes the schema name. The omitted schema name can be changed in the application program. The schema name specified in the SET SCHEMA statement is for the prepared statements of PREPARE and EXECUTE IMMEDIATE statements. The changed schema name is valid until the session terminates.

The following is an example of specifying the SET SCHEMA statement:

Example:

In this example, the schema name is changed to "SCH1".

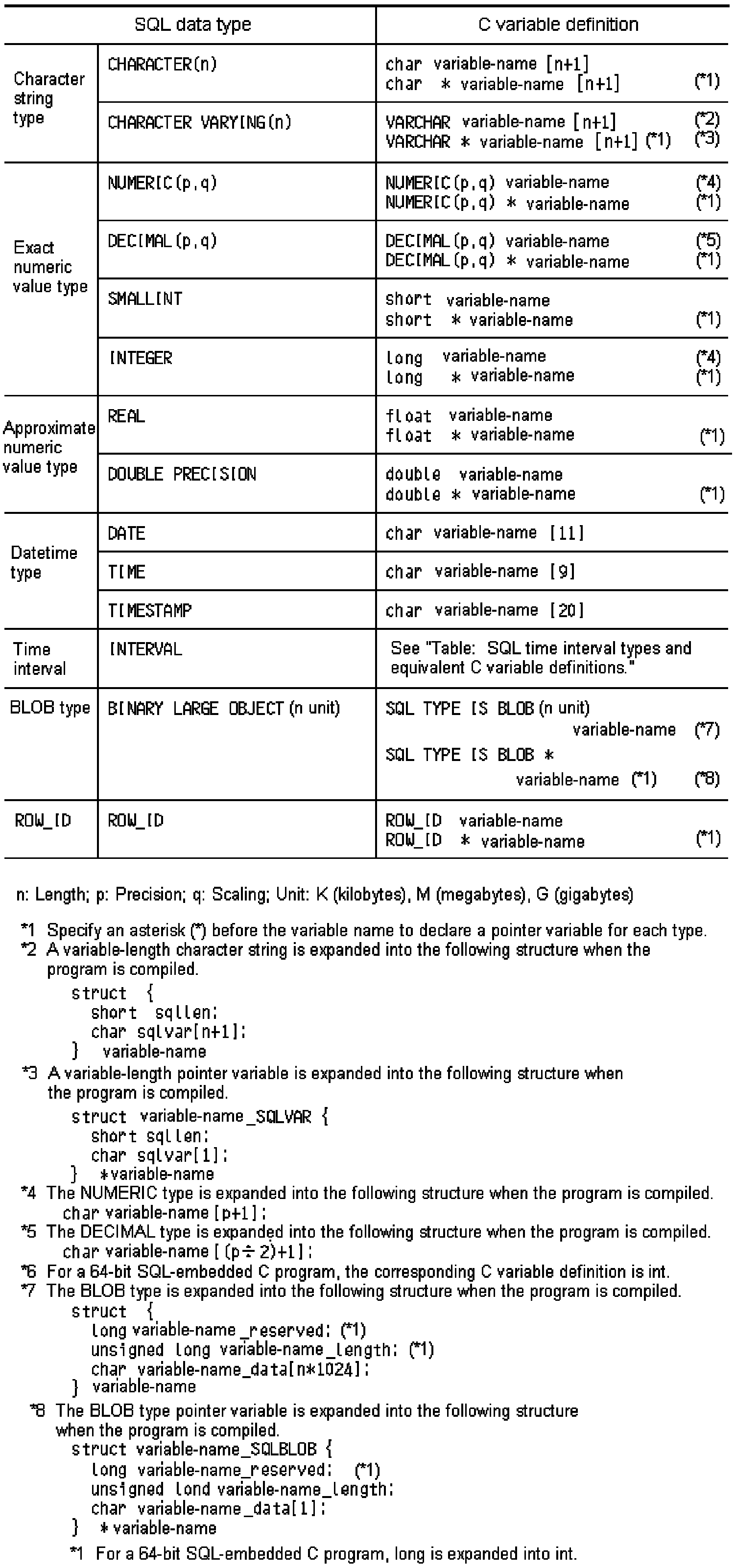


```
                 ⋮
EXEC SQL CONNECT TO 'RDBDB1' USER 'USER1/PASS1';
                                    Login name

strcpy (str.sqlvar, "SELECT * FROM TBL");          ....... Schema name omitted
                 ⋮
EXEC SQL PREPARE STM FROM : str;                   ....... Schema name to be referenced:  USER1
EXEC SQL SET SCHEMA 'SCH1'                         ....... Schema name changed
EXEC SQL PREPARE STM FROM : str;                   ....... Schema name to be referenced:  SCH1
                 ⋮
```

Appendix A SQL Data types and equivalent Host Variable Data Types

Table: SQL data types and equivalent C language host variable data types lists SQL data types and the equivalent C language host variable data types.

[Table: SQL data types and equivalent C language host variable data types]



SQL data type		C variable definition
Character string type	CHARACTER(n)	char variable-name [n+1] char * variable-name [n+1] (*1)
	CHARACTER VARYING(n)	VARCHAR variable-name [n+1] (*2) VARCHAR * variable-name [n+1] (*1) (*3)
Exact numeric value type	NUMERIC(p,q)	NUMERIC(p,q) variable-name (*4) NUMERIC(p,q) * variable-name (*1)
	DECIMAL(p,q)	DECIMAL(p,q) variable-name (*5) DECIMAL(p,q) * variable-name (*1)
	SMALLINT	short variable-name short * variable-name (*1)
	INTEGER	long variable-name (*4) long * variable-name (*1)
Approximate numeric value type	REAL	float variable-name float * variable-name (*1)
	DOUBLE PRECISION	double variable-name double * variable-name (*1)
Datetime type	DATE	char variable-name [11]
	TIME	char variable-name [9]
	TIMESTAMP	char variable-name [20]
Time interval	INTERVAL	See "Table: SQL time interval types and equivalent C variable definitions."
BLOB type	BINARY LARGE OBJECT (n unit)	SQL TYPE IS BLOB (n unit) variable-name (*7) SQL TYPE IS BLOB * variable-name (*1) (*8)
ROW_ID	ROW_ID	ROW_ID variable-name ROW_ID * variable-name (*1)

n: Length; p: Precision; q: Scaling; Unit: K (kilobytes), M (megabytes), G (gigabytes)

*1 Specify an asterisk (*) before the variable name to declare a pointer variable for each type.

*2 A variable-length character string is expanded into the following structure when the program is compiled.

```
struct  {
   short  sqllen;
   char sqlvar[n+1];
}   variable-name
```

*3 A variable-length pointer variable is expanded into the following structure when the program is compiled.

```
struct variable-name_SQLVAR {
   short sqllen;
   char sqlvar[1];
}   *variable-name
```

*4 The NUMERIC type is expanded into the following structure when the program is compiled.

```
char variable-name [p+1];
```

*5 The DECIMAL type is expanded into the following structure when the program is compiled.

```
char variable-name [(p÷2)+1];
```

*6 For a 64-bit SQL-embedded C program, the corresponding C variable definition is int.

*7 The BLOB type is expanded into the following structure when the program is compiled.

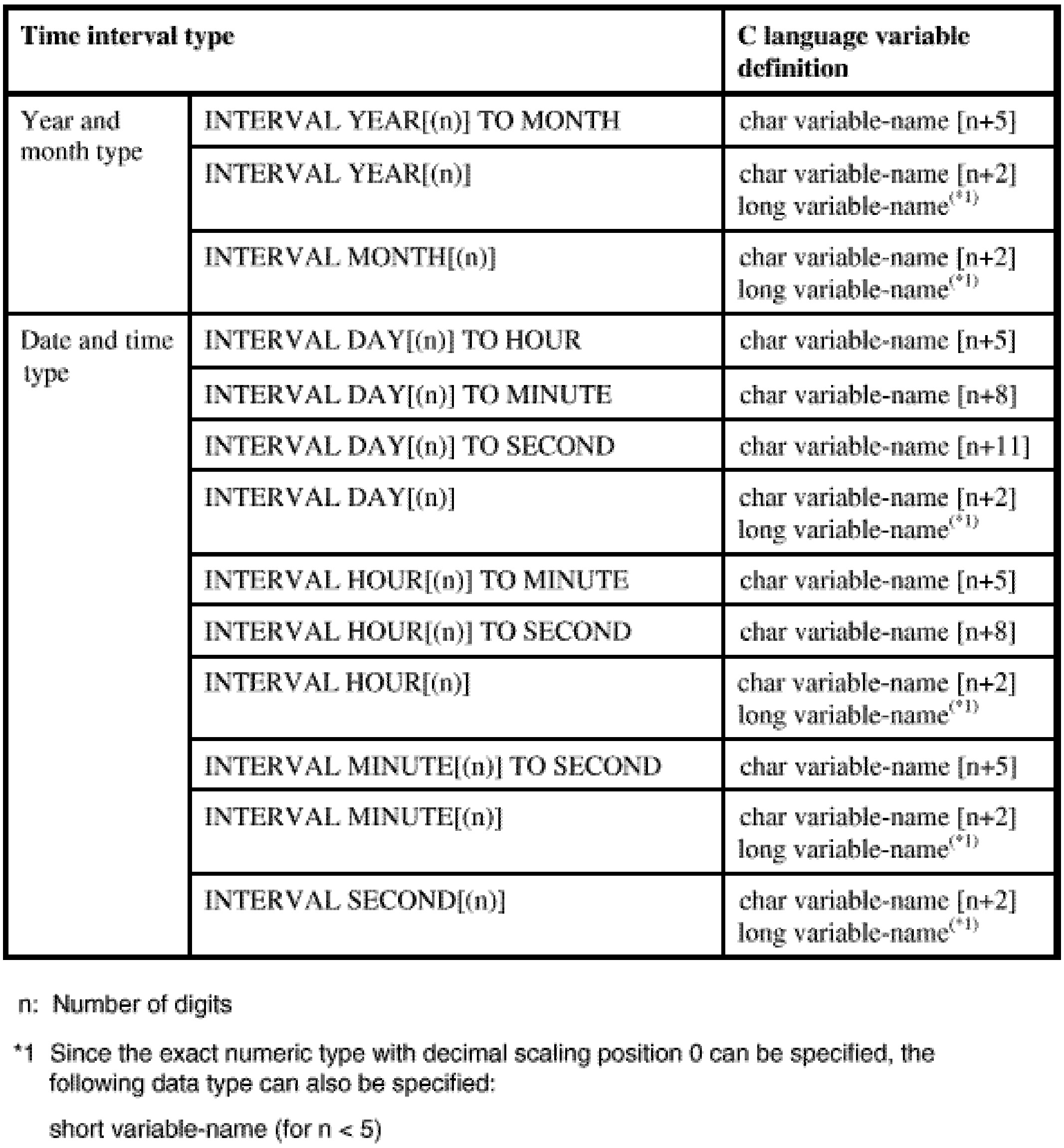
```
struct  {
   long variable-name_reserved; (*1)
   unsigned long variable-name_length; (*1)
   char variable-name_data[n*1024];
}  variable-name
```

*8 The BLOB type pointer variable is expanded into the following structure when the program is compiled.

```
struct variable-name_SQLBLOB {
   long variable-name_reserved;   (*1)
   unsigned lond variable-name_length;
   char variable-name_data[1];
}   * variable-name
```

*1 For a 64-bit SQL-embedded C program, long is expanded into int.

[Table: Time interval types and equivalent data types for C language host variables]



Time interval type		C language variable definition
Year and month type	INTERVAL YEAR[(n)] TO MONTH	char variable-name [n+5]
	INTERVAL YEAR[(n)]	char variable-name [n+2] long variable-name(*1)
	INTERVAL MONTH[(n)]	char variable-name [n+2] long variable-name(*1)
Date and time type	INTERVAL DAY[(n)] TO HOUR	char variable-name [n+5]
	INTERVAL DAY[(n)] TO MINUTE	char variable-name [n+8]
	INTERVAL DAY[(n)] TO SECOND	char variable-name [n+11]
	INTERVAL DAY[(n)]	char variable-name [n+2] long variable-name(*1)
	INTERVAL HOUR[(n)] TO MINUTE	char variable-name [n+5]
	INTERVAL HOUR[(n)] TO SECOND	char variable-name [n+8]
	INTERVAL HOUR[(n)]	char variable-name [n+2] long variable-name(*1)
	INTERVAL MINUTE[(n)] TO SECOND	char variable-name [n+5]
	INTERVAL MINUTE[(n)]	char variable-name [n+2] long variable-name(*1)
	INTERVAL SECOND[(n)]	char variable-name [n+2] long variable-name(*1)

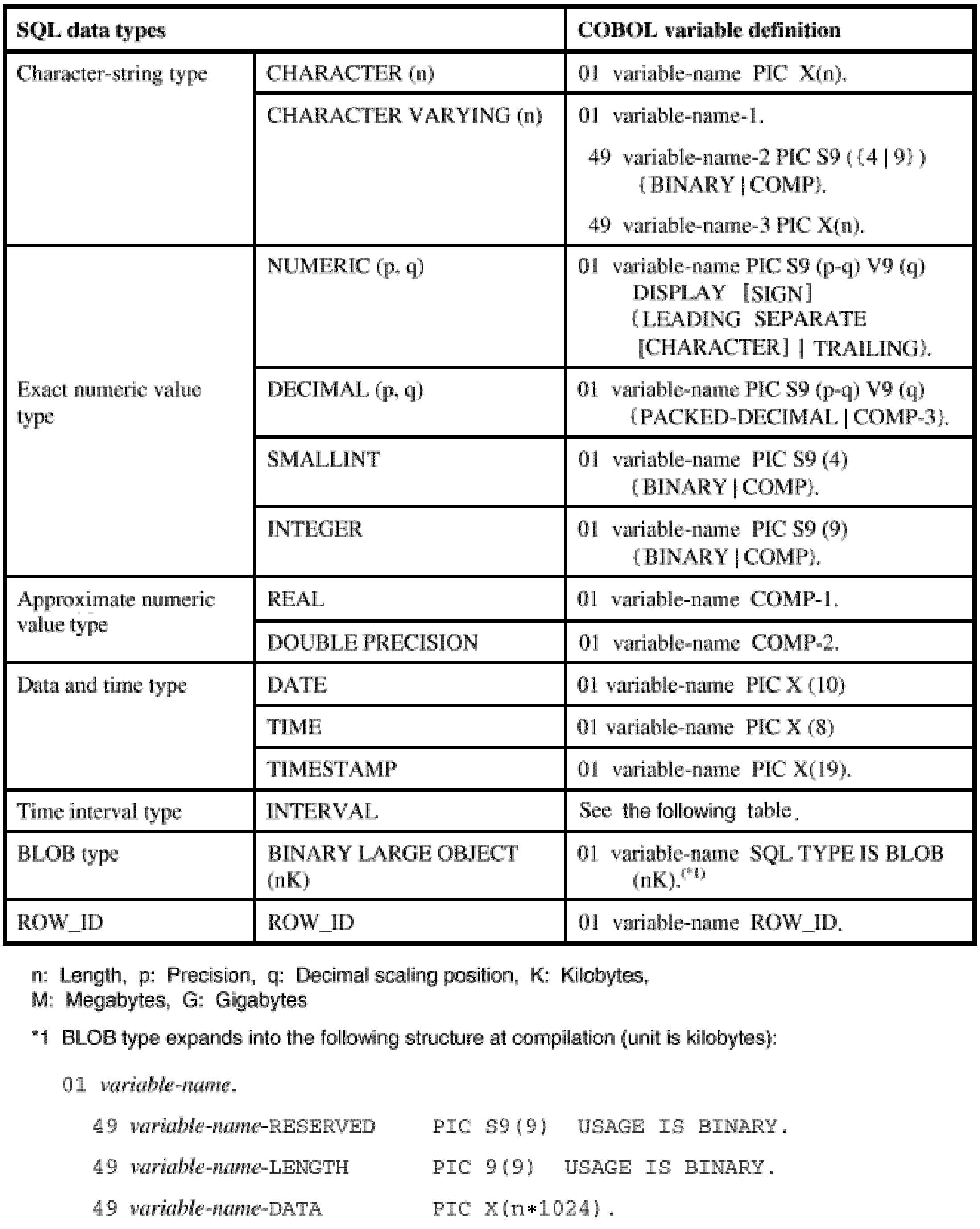
n: Number of digits

*1 Since the exact numeric type with decimal scaling position 0 can be specified, the following data type can also be specified:

short variable-name (for n < 5)

Table: SQL data types and equivalent data types for COBOL host variables lists the data types and the equivalent data types for COBOL host variables.

[Table: SQL data types and equivalent data types for COBOL host variables]



SQL data types		COBOL variable definition
Character-string type	CHARACTER (n)	01 variable-name PIC X(n).
	CHARACTER VARYING (n)	01 variable-name-1. 49 variable-name-2 PIC S9 ({4 | 9}) {BINARY | COMP}. 49 variable-name-3 PIC X(n).
Exact numeric value type	NUMERIC (p, q)	01 variable-name PIC S9 (p-q) V9 (q) DISPLAY [SIGN] {LEADING SEPARATE [CHARACTER] | TRAILING}.
	DECIMAL (p, q)	01 variable-name PIC S9 (p-q) V9 (q) {PACKED-DECIMAL | COMP-3}.
	SMALLINT	01 variable-name PIC S9 (4) {BINARY | COMP}.
	INTEGER	01 variable-name PIC S9 (9) {BINARY | COMP}.
Approximate numeric value type	REAL	01 variable-name COMP-1.
	DOUBLE PRECISION	01 variable-name COMP-2.
Data and time type	DATE	01 variable-name PIC X (10)
	TIME	01 variable-name PIC X (8)
	TIMESTAMP	01 variable-name PIC X(19).
Time interval type	INTERVAL	See the following table.
BLOB type	BINARY LARGE OBJECT (nK)	01 variable-name SQL TYPE IS BLOB (nK).(*1)
ROW_ID	ROW_ID	01 variable-name ROW_ID.

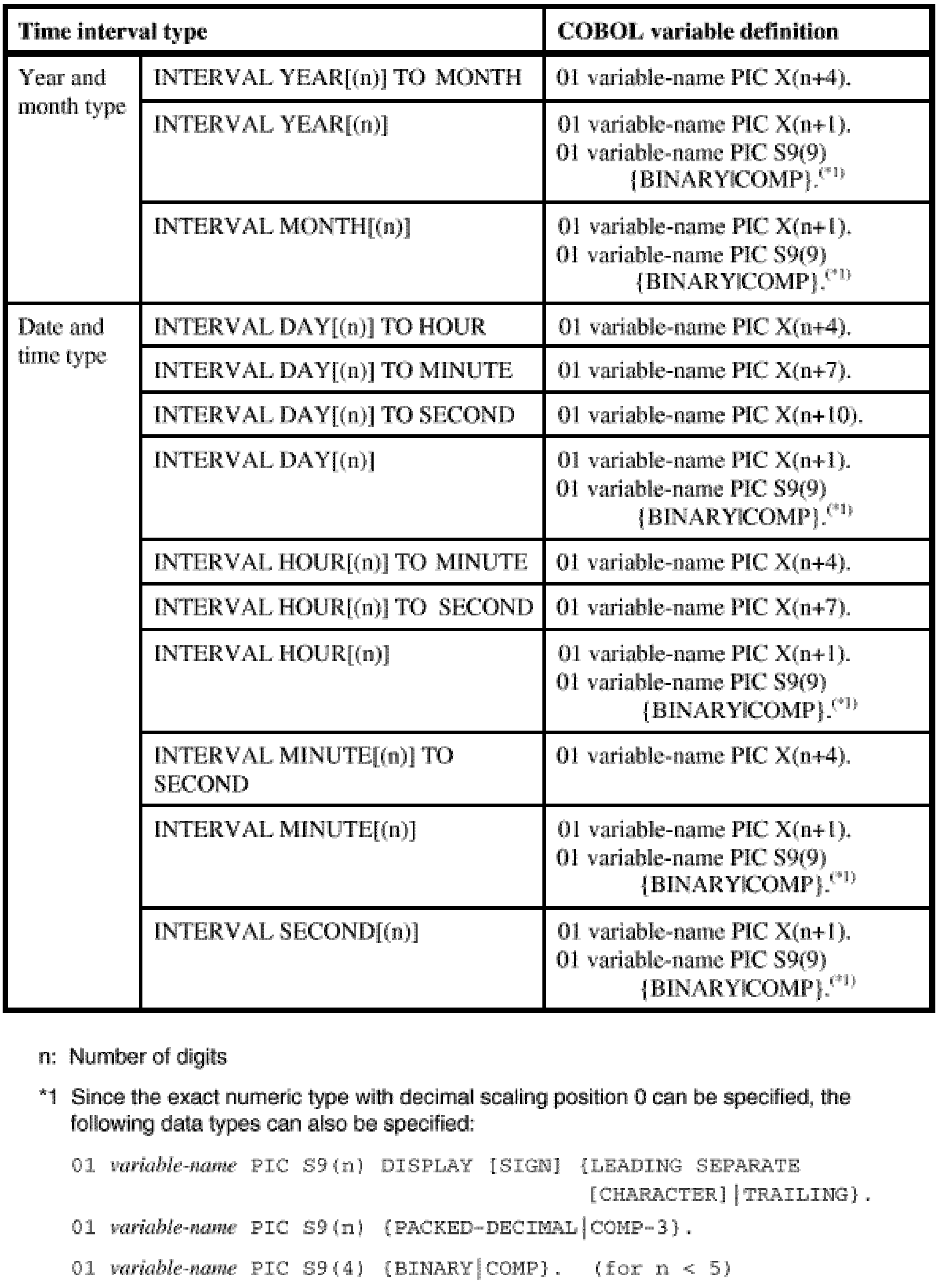
n: Length, p: Precision, q: Decimal scaling position, K: Kilobytes,
M: Megabytes, G: Gigabytes

*1 BLOB type expands into the following structure at compilation (unit is kilobytes):

```
01 variable-name.
   49 variable-name-RESERVED     PIC S9(9)  USAGE IS BINARY.
   49 variable-name-LENGTH       PIC 9(9)  USAGE IS BINARY.
   49 variable-name-DATA         PIC X(n*1024).
```

Table: Time interval types and equivalent data types for COBOL host variables shows the time interval types and equivalent data types for COBOL host variables.

[Table: Time interval types and equivalent data types for COBOL host variables]



Time interval type		COBOL variable definition
Year and month type	INTERVAL YEAR[(n)] TO MONTH	01 variable-name PIC X(n+4).
	INTERVAL YEAR[(n)]	01 variable-name PIC X(n+1). 01 variable-name PIC S9(9) {BINARY|COMP}.(*1)
	INTERVAL MONTH[(n)]	01 variable-name PIC X(n+1). 01 variable-name PIC S9(9) {BINARY|COMP}.(*1)
Date and time type	INTERVAL DAY[(n)] TO HOUR	01 variable-name PIC X(n+4).
	INTERVAL DAY[(n)] TO MINUTE	01 variable-name PIC X(n+7).
	INTERVAL DAY[(n)] TO SECOND	01 variable-name PIC X(n+10).
	INTERVAL DAY[(n)]	01 variable-name PIC X(n+1). 01 variable-name PIC S9(9) {BINARY|COMP}.(*1)
	INTERVAL HOUR[(n)] TO MINUTE	01 variable-name PIC X(n+4).
	INTERVAL HOUR[(n)] TO SECOND	01 variable-name PIC X(n+7).
	INTERVAL HOUR[(n)]	01 variable-name PIC X(n+1). 01 variable-name PIC S9(9) {BINARY|COMP}.(*1)
	INTERVAL MINUTE[(n)] TO SECOND	01 variable-name PIC X(n+4).
	INTERVAL MINUTE[(n)]	01 variable-name PIC X(n+1). 01 variable-name PIC S9(9) {BINARY|COMP}.(*1)
	INTERVAL SECOND[(n)]	01 variable-name PIC X(n+1). 01 variable-name PIC S9(9) {BINARY|COMP}.(*1)

n: Number of digits

*1 Since the exact numeric type with decimal scaling position 0 can be specified, the following data types can also be specified:

```
01 variable-name PIC S9(n) DISPLAY [SIGN] {LEADING SEPARATE
                                    [CHARACTER]|TRAILING}.
01 variable-name PIC S9(n) {PACKED-DECIMAL|COMP-3}.
01 variable-name PIC S9(4) {BINARY|COMP}.  (for n < 5)
```

Appendix B Handling RDB Messages

User handling of some messages can be referenced online. These messages are issued when RDB commands are being executed or an application program is being compiled. These messages also include messages set in a message variable (SQLMSG) of an application program.

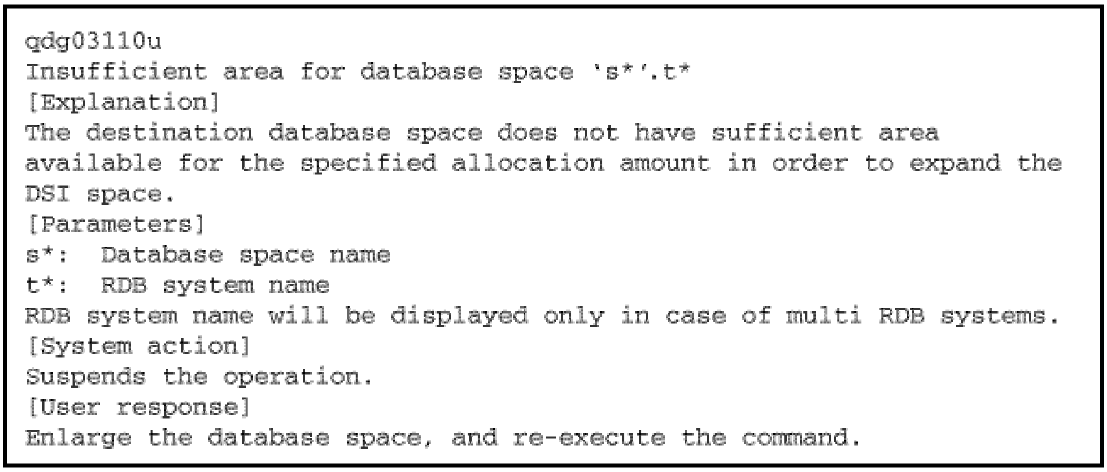
Use the online manual SymfoWARE/RDB Message Reference to reference user handling of messages issued by the SymfoWARE/RDB.

This appendix describes reference methods for each task.

■When executing an RDB command

Example 1:

Display a description of the qdg03110u message issued by executing an RDB command.

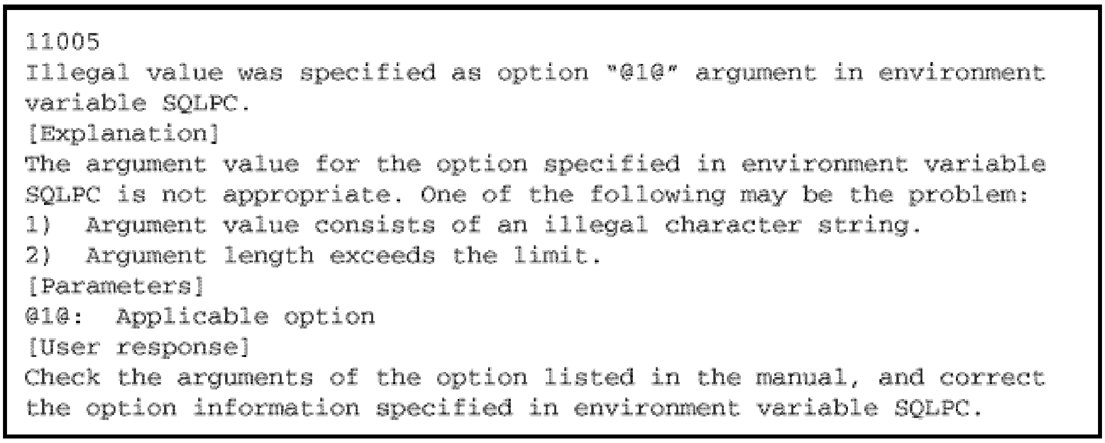


```
qdg03110u
Insufficient area for database space 's*'.t*
[Explanation]
The destination database space does not have sufficient area
available for the specified allocation amount in order to expand the
DSI space.
[Parameters]
s*:  Database space name
t*:  RDB system name
RDB system name will be displayed only in case of multi RDB systems.
[System action]
Suspends the operation.
[User response]
Enlarge the database space, and re-execute the command.
```

■When compiling an application program (C language)

Example 2:

Display a description of message 11005 issued by the C precompiler.

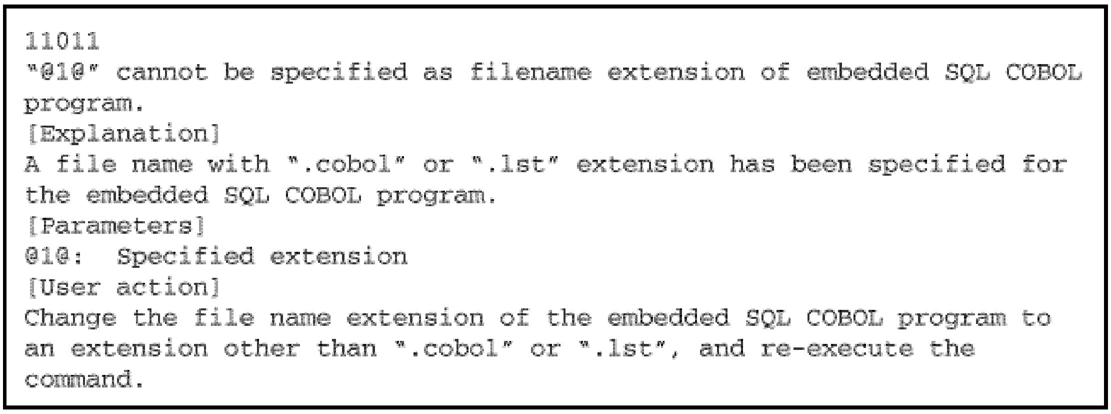


```
11005
Illegal value was specified as option "@1@" argument in environment
variable SQLPC.
[Explanation]
The argument value for the option specified in environment variable
SQLPC is not appropriate. One of the following may be the problem:
1)  Argument value consists of an illegal character string.
2)  Argument length exceeds the limit.
[Parameters]
@1@:  Applicable option
[User response]
Check the arguments of the option listed in the manual, and correct
the option information specified in environment variable SQLPC.
```

■When compiling an application program (COBOL)

Example 3:

Display a description of message 11011 issued by the COBOL precompiler.

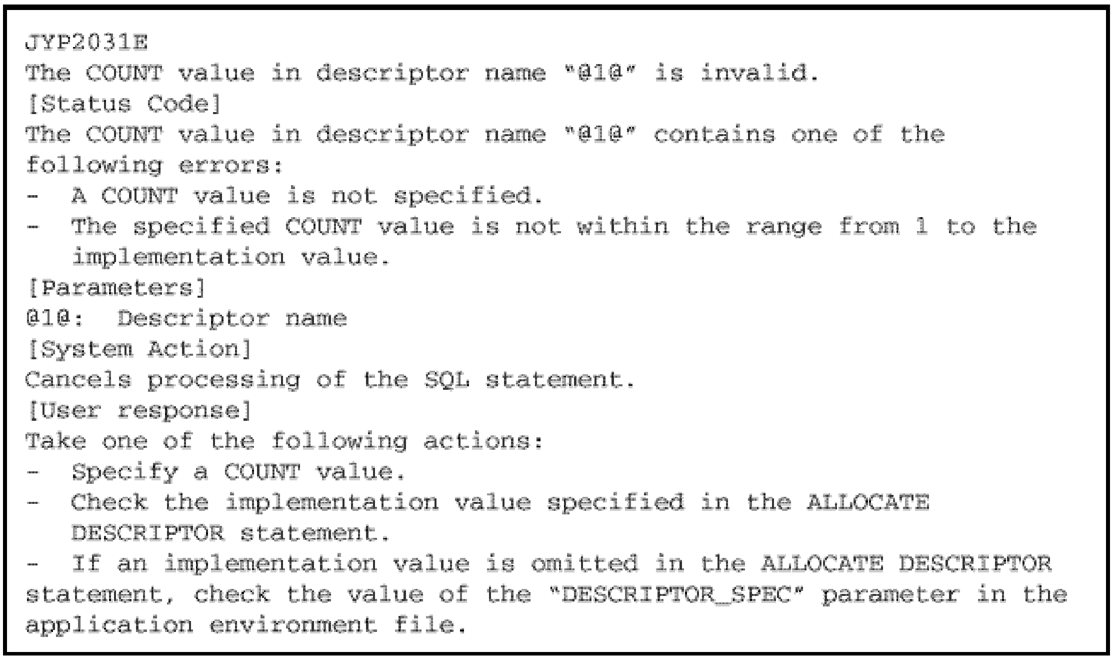


```
11011
"@1@" cannot be specified as filename extension of embedded SQL COBOL
program.
[Explanation]
A file name with ".cobol" or ".lst" extension has been specified for
the embedded SQL COBOL program.
[Parameters]
@1@:  Specified extension
[User action]
Change the file name extension of the embedded SQL COBOL program to
an extension other than ".cobol" or ".lst", and re-execute the
command.
```

■When executing an application program

Example 4:

Display a description of message JYP2031E set in the message variable (SQLMSG) when executing an application program.



```
JYP2031E
The COUNT value in descriptor name "@1@" is invalid.
[Status Code]
The COUNT value in descriptor name "@1@" contains one of the
following errors:
-  A COUNT value is not specified.
-  The specified COUNT value is not within the range from 1 to the
   implementation value.
[Parameters]
@1@:  Descriptor name
[System Action]
Cancels processing of the SQL statement.
[User response]
Take one of the following actions:
-  Specify a COUNT value.
-  Check the implementation value specified in the ALLOCATE
   DESCRIPTOR statement.
-  If an implementation value is omitted in the ALLOCATE DESCRIPTOR
statement, check the value of the "DESCRIPTOR_SPEC" parameter in the
application environment file.
```

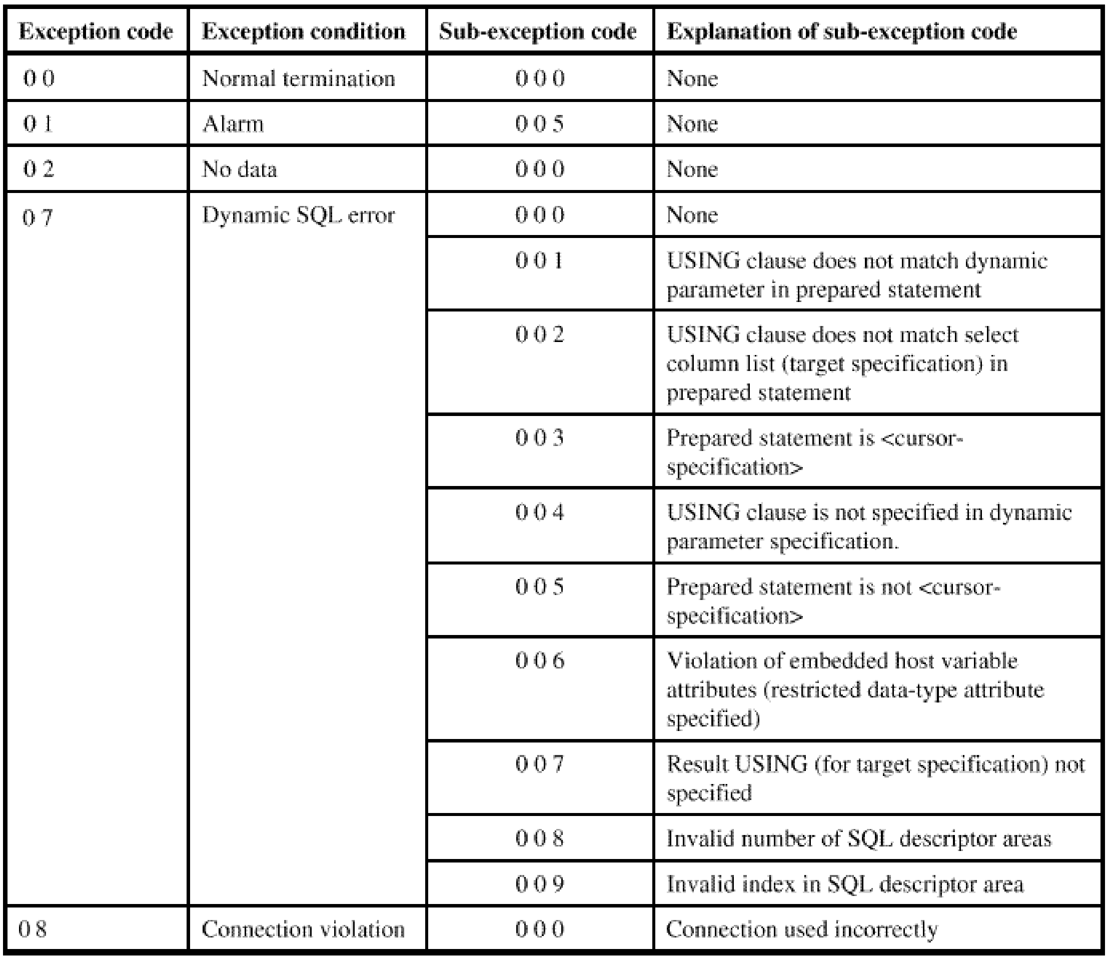
Appendix C SQLSTATE Values

The system reports the execution results of SQL statements in status variable SQLSTATE while an application program is running. The application program checks SQLSTATE whenever an SQL statement is to be executed, and processes SQL information while checking the result of SymfoWARE/RDB processing.

■SQLSTATE

Status variable SQLSTATE is a 5-digit character-type variable. The value set in SQLSTATE, the result of an SQL statement being executed, is known as the status code. The status code is made up of two parts: a 2-digit exception code, followed by a 3-digit sub-exception code. Table: SQLSTATE values lists the code values for the execution results of SQL statements.

[Table: SQLSTATE values]



Exception code	Exception condition	Sub-exception code	Explanation of sub-exception code
0 0	Normal termination	0 0 0	None
0 1	Alarm	0 0 5	None
0 2	No data	0 0 0	None
0 7	Dynamic SQL error	0 0 0	None
		0 0 1	USING clause does not match dynamic parameter in prepared statement
		0 0 2	USING clause does not match select column list (target specification) in prepared statement
		0 0 3	Prepared statement is <cursor-specification>
		0 0 4	USING clause is not specified in dynamic parameter specification.
		0 0 5	Prepared statement is not <cursor-specification>
		0 0 6	Violation of embedded host variable attributes (restricted data-type attribute specified)
		0 0 7	Result USING (for target specification) not specified
		0 0 8	Invalid number of SQL descriptor areas
		0 0 9	Invalid index in SQL descriptor area
0 8	Connection violation	0 0 0	Connection used incorrectly

Exception code	Exception condition	Sub-exception code	Explanation of sub-exception code
0 8	Connection violation	0 0 2	Connection being used
		0 0 3	Connection does not exist
		0 0 4	SQL server refused SQL connection
0 9	Triggered operation exception	0 0 0	None
0 A	Unsupported function	0 0 0	None
2 1	Radix violation	0 0 0	Execution of a single row SELECT statement resulted in retrieval of two or more rows. Alternatively, the result of a subquery included directly in a comparison predicate was the retrieval of two or more rows
2 2	Data exception	0 0 1	Right truncation of character data
		0 0 2	Indicator variable that sets null value not specified
		0 0 3	Range of numerics that can be handled exceeded
		0 0 5	Violation of assignment rule
		0 0 8	Range of date and time that can be handled exceeded
		0 1 1	Substring specified incorrectly
		0 1 2	Division using 0 as divisor
		0 1 5	Range of time interval that can be handled exceeded
		0 1 8	Format of character string for type conversion incorrect
		0 1 9	Invalid escape character
		0 2 4	Incorrect character-type host variable (null character does not exist in definition size for input host variable)
		0 2 7	TRIM character specified incorrectly
		7 0 1	Incorrect variable-length character string (value of length part less than 0 or defined size exceeded)
		7 0 2	Access outside range of DSI
		7 0 3	Incorrect numeric (value of fixed-point sign part invalid)

Exception code	Exception condition	Sub-exception code	Explanation of sub-exception code
22	Data exception	7 0 4	Incorrect character (value of national-language character string invalid)
2 3	Violation of NOT NULL constraint or unique constraint	0 0 0	None
2 4	Invalid cursor status	0 0 0	None
2 5	Invalid transaction status	0 0 0	None
2 6	Invalid SQL statement identifier	0 0 0	None
2 8	Invalid authorization specification	0 0 0	None
2 B	The dependent privilege descriptor still exists. (*1)	0 0 0	None
2 E	Invalid connection name	0 0 0	None
2 F	Routine calling error	0 0 0	None
		0 0 3	Execution of prohibited SQL statement
3 3	Invalid descriptor name	0 0 0	None
3 4	Invalid cursor name	0 0 0	None
3 7	SQL statement syntax error	0 0 0	None
3 D	Invalid catalog name	0 0 0	None
3 F	Invalid schema name	0 0 0	None
4 0 6	Transaction rollback	0 0 1	Failure or realtime exceeded during transaction serialization
		0 0 3	Statement end unclear (processing time over or processing is forcibly terminated)
		7 0 1	Insufficient temporary log area
		7 0 3	The node being accessed has gone down.
		7 0 2	COMMIT execution error
		8 0 1	DDL statement processing error
4 2	Violation of access rule	0 0 0	No authority
4 4	Violation of WITH CHECK option during updating of view	0 0 0	None
6 0	User transmission exception	xxx	Exception sent by the user (*3)
7 0	Quantitative limit	0 0 0	None

Exception code	Exception condition	Sub-exception code	Explanation of sub-exception code
7 1	Environment incompatibility or operating conflict	0 0 0	Operating system limit value exceeded
		0 0 1	SymfoWARE/RDB not started
		0 0 2	Server system not started
		003	An access request to the failed node
		0 1 0	Invalid environment variable or operating environment file format
		0 2 0	SymfoWARE/RDB limit value exceeded
		0 3 0	SymfoWARE/RDB environment conflict
		031	Unsuccessful loading of code conversion library
		2 1 0	DSI undefined or not selected
		2 2 0	DSI unformatted
		3 0 0	Cursor conflict
		4 0 0	Usage rules (*1)
		4 1 0	Access inhibit
		5 0 0	Database or DSO undefined
		6 0 0	Session unlockable status
		7 0 0	Execution multiplicity for transactions exceeded
		7 1 0	Status requiring recovery
		7 2 0	Data base or dictionary conflict
		7 3 0	DDL statement processing error
		7 4 0	Split transfer conflict
		7 5 0	Temporary table limit value exceeded
		8 0 0	DSI automatic capacity expansion error
		900	Error in hot-standby environment
		A00	Violation of sequence used
7 2	Insufficient area (such as in external storage area)	0 0 0	Insufficient DSI storage area
		0 1 0	Insufficient database space
		1 0 0	Insufficient directory area
		2 0 0	Insufficient work table area
7 3	Insufficient memory area	1 0 0	Number of local memory pools exceeds specified value of upper limit
		1 1 0	Insufficient local memory
		2 0 0	Number of shared memory pools exceeds specified value of upper limit

Exception code	Exception condition	Sub-exception code	Explanation of sub-exception code
73	Insufficient memory area	210	Insufficient shared memory
		300	Insufficient shared buffers
74	Error in I-O to external storage area	000	Database space I-O error
		010	No access authority to database space
		100	I-O error to directory area
		110	No access authority to directory area
		200	Work file I-O error
		210	No access authority to work file
		400	Temporary log area I-O error
		410	No access authority to temporary log area
		500	Error in I-O to snap file
		600	Other file input-output error
		610	No access authority
		720	Dictionary input-output error
75	Connection environment error	000	Connection execution environment error
76	System environment failure	000	System failure
77	Operational error by another product	000	Error detection in Power AIM/T JNL

*1: The privilege descriptor consists of the following items:
- The resources (such as tables and views) that the privilege receiver owns
- The authority that the privilege receiver assigns to another authorization identifier

Remarks: If the table has no code that works in conjunction with the ODBC-based PC tools, refer to "MSDN Library Visual Studio."

*2 This can be set even when the rdbfmt command is being executed for the DSI to be processed.

*3 Any characters from 0 to 9 and A to Z that are set in the sub-exception code of the SIGNAL statement are represented as three digits.

Remarks: If the code does not exist when a personal computer tool that supports ODBC is linked, see the MSDN Library Visual Studio.

Glossary

ALL set function

The ALL set function is a set function for which ALL is specified in the argument. Rows containing the same value are targeted by the ALL set function.

Related term: DISTINCT set function

Application program

In general, programs used in the work of a computer user are called application programs. In this manual, a program that reads data from a database and writes data to a database for user work processing is called an application program.

Arbitrary character specifier

In the LIKE predicate of an SQL data manipulation statement, the arbitrary character specifier specifies that a data position can contain any character. Related terms: escape character and arbitrary string specifier

---

Arbitrary string specifier

In the LIKE predicate of an SQL data manipulation statement, the arbitrary string specifier specifies that a data position can contain any character string of zero or more characters.

Related terms: escape character and arbitrary character specifier

---

Base table

A table defined as a base table in logical structure definition. The data body is stored in a database space.

BETWEEN predicate

In a data manipulation SQL statement, the BETWEEN predicate is the search condition used to specify a range comparison in the format A BETWEEN B AND C (B?? A?? C).

Related term: predicate

Boolean operator

Boolean operators AND, OR, and NOT are operators that specify logical operations in search conditions. In SQL search conditions, the logical operators logical conjunction (AND) or logical disjunction (OR) can be performed between predicates, or negation (NOT) can be performed within a predicate.

Column

A constituent element of a table. A relational database represents data using two-dimensional tables consisting of rows and columns.

Column name (item name)

A column name is the name assigned to a column, defined in the schema definition. The column name is used in SQL statements for data manipulation to specify a column to be processed.

Commit

Makes the data manipulation of a transaction being processed take effect. The data manipulation in the transaction is physically reflected in the database. An application program deliberately controls a commit by issuing a COMMIT statement.

Related term: rollback

Comparison predicate

The comparison predicate is a search condition specified using the comparison operators =, <, <=, >, >=, and <>.

Connection

Connection refers to the relationship that connects a client to a server. A connection is made by specifying a CONNECT statement in an application program. Conversely, a DISCONNECT statement cuts off the connection.

Correlation name

Correlation names are table names used by SQL statements for data manipulation to manipulate data. A correlation name is valid only for the SQL statement for which it is specified.

Cursor

A cursor is an indicator that indicates a row to be manipulated. A cursor is defined by a cursor declaration statement. The OPEN and CLOSE statements start and stop cursor operations, respectively. The FETCH statement moves the cursor.

Cursor SQL statement

A cursor SQL statement is a data manipulation SQL statement that uses a cursor to specify rows to be manipulated.

Data manipulation SQL

A data manipulation SQL is an SQL statement used to reference, add, delete, or update a data base.

Database name

Many databases can be created as units of administration and design on one server system. To identify each database uniquely, each is assigned a unique name (database name) on the server system.

DEFAULT clause

An element of the definition of a column in a table. If the following condition applies, the value defined in the DEFAULT clause is inserted. The data to be inserted in a column is not specified when a row is inserted in a table using the INSERT statement. If a DEFAULT clause is not defined for a column, a NULL value is inserted.

DELETE statement

A data manipulation SQL statement used to delete a row from a table.

DISTINCT set function

The DISTINCT set function is a set function for which DISTINCT is specified in the set function argument. Rows containing the same values are not removed for a DISTINCT set function.

Related term: ALL set function

DS/90 7000 series

A UNIX computer from Fujitsu Limited and PFU Limited that uses the SuperSPARC high-performance RISC processor.

Dynamic SQL

Dynamic SQL is a function for generating and executing SQL statements when executing an application program. In general, this function is used by general-purpose package programs.

Embedded exception declaration

An embedded exception declaration specifies the processing to be performed if an exception condition is issued when an SQL statement is executed in an application program. The WHENEVER statement is used to specify an embedded exception declaration.

Embedded SQL

SQL statements can be used as embedded SQL statements. Use embedded SQL statements to manipulate data in application programs written in high-level languages such as C language or COBOL. For example, when COBOL is used, specify an SQL statement embedded between EXEC SQL and END-EXEC.

Escape character

An escape character is an alternate character for underscore (_) of an arbitrary character specifier or for percent symbol (%)of an arbitrary string specifier. An escape character is specified in a LIKE predicate. A LIKE predicate is used to specify a retrieval condition for data manipulation. However, to retrieve the actual underscore or percent symbol, an alternate character, the escape character, must be defined as the arbitrary character specifier for these symbols.

Related term: arbitrary character specifier and arbitrary string specifier

Exception condition

When an SQL statement is executed in an application program, data to be processed may not be able to be found or an error may occur. Such a condition is known as an exception condition or exception. If an exception condition occurs while an application program is running, the status code corresponding to the exception condition is set in SQLSTATE.

EXISTS predicate

The EXISTS predicate specifies for an SQL statement the search condition that rows are manipulated depending on the result (true or false) of the specified subquery. The EXISTS predicate is specified in the EXISTS subquery.

Related term: predicate

Function

The functions that can be used in SQL statements are set functions.

Related term: set function

Grouped column

A grouped column is a column specified in the GROUP BY clause of a table expression for a query specification.

Related term: grouped table

Grouped table

A grouped table is a table derived when a GROUP BY clause is specified in a table expression for a query specification. Rows containing the same grouped column value are grouped in a grouped table. Values that can be fetched from a grouped table are grouped column values or the values resulting from a set function being used for a group.

Related term: grouped column

Host variable

A variable for passing data between an application program and a database in SQL statements that manipulate data.

IN predicate

In the search condition specification for an SQL statement, the IN predicate specifies that rows are manipulated depending on comparisons with a set of values. The IN predicate is the predicate in IN (C, D, ...) or IN subquery.

Related term: predicate

Indicator variable

In high-level languages such as C language and COBOL, the indicator variable is a variable specified to be paired with an SQL data variable. Indicator variables are used when SQL statements are used to fetch data from, and update a database. When SQL statements are used to update a database, the indicator variable indicates whether or not data stored in the SQL data variable contains a null value. When SQL statements are used to reference a database, the indicator variable indicates whether or not the execution result of the SQL statement has a null value stored in the SQL data variable. The indicator variable also shows the number of characters in character-string data stored in the SQL data variable.

INSERT statement

A data manipulation SQL statement used to add a row to a table.

Join

A type of relation operation. In a relational database, one table can be made from two or more tables. This operation is called join.

LIKE predicate

In the search condition specification for an SQL statement, the LIKE predicate specifies that rows are manipulated depending on a comparison of character type data. The LIKE predicate is the predicate in A LIKE B.

Related term: predicate

Logical structure

This is one of the structures of a database along with the storage structure and physical structure. The data structure that includes the schema, table and column configuration, and column data types is called the logical structure. Constrains such as unique constrains, authorities, procedure routines, and triggers are also elements of the logical structure.

Related terms: storage structure and physical structure

Logical structure definition

An element of SymfoWARE/RDB database definition (also called schema definition). Table and view table definitions apply to logical structure definition.

Non-cursor SQL statement

An SQL statement used for data manipulation, the non-cursor SQL statement does not use a cursor to specify rows to be manipulated. Instead, the rows to be processed are specified in the search condition specified in the SQL statement.

NOT NULL constraint

The NOT NULL constraint is a constraint that can be applied to columns of tables. The NOT NULL constraint means that the value in a row of a column of a table cannot be a NULL value.

Null (NULL)

Null is the state in which the value in a row of a column of a table is undefined.

NULL predicate

In the search condition specification for an SQL statement, the NULL predicate specifies that rows are manipulated depending on whether the column value is null. The NULL predicate is the predicate in A IS NULL.

Related term: predicate

Parameter name

A parameter name is the name of a parameter variable.

Related term: Parameter variable

Parameter variable

Variable used to set an argument value when a procedure routine is called.

Physical structure

One of the database structures along with the logical structure and storage structure. The database space located on a magnetic disk volume is called the physical structure. A database space is an aggregate of fixed-size blocks.

Related terms: storage structure and logical structure

Physical structure definition

An element of SymfoWARE/RDB database definition for creating a database space.

Predicate

A predicate is an element in SQL statements used to specify search conditions for rows to be manipulated. Seven types of predicate are available: comparison, BETWEEN, IN, LIKE, NULL, quantified, and EXISTS. Search conditions are specified using a predicate and Boolean operators.

Procedure routine

A procedure routine defines processing procedures for a database using SQL.

Procedure routine definition

A procedure routine definition defines a procedure routine using a logical structure definition.

Related term: Procedure routine

Quantified predicate

Using a quantified predicate is one method of specifying an SQL search condition. A quantified predicate uses a quantifier (ALL, SOME, or ANY) to specify the comparison of the object rows with a group of columns in the same column.

Quantified value list

A quantified value list is used when the IN predicate specified in the SQL statement search condition specifies rows to be manipulated based on comparison with a set of values. The quantified value list specifies the set of values to be compared by the IN predicate. For example, to specify rows for which the value in column C1 is 1, 3, or 5, specify C1 IN (1, 3, 5). The part enclosed in parentheses is the quantified value list.

Query expression

A query expression is specified in a cursor declaration to derive a table. A query specification, or several query specifications joined by UNION, can be specified in a query expression.

Query specification

A query specification derives a table from an INSERT statement of an SQL statement or from a query expression. A query specification consists of a table expression that specifies the rows to be derived from the original table and a select column list that specifies the required columns from these rows.

Read-only cursor

In an SQL data manipulation statement, a cursor that cannot be used to update and delete is called a read-only cursor. The cursor declaration specifies whether a cursor is read-only. For example, if two or more tables are specified in the FROM clause of a query specification, the cursor is a read-only cursor.

Related term: updatable cursor

Relation operation

With relational databases, a relation operation allows only data from specified columns or a collection of data that meets certain conditions to be fetched. Three types of relation operations are available: selection, projection, and join. Selection refers to fetching rows that match specified conditions from a table. Projection refers to fetching only specified columns from a table. Join refers to specifying join conditions to create one table from two or more tables.

Rollback

Nullifying the data manipulation of a transaction that is being processed. The two kinds of rollback can be performed. Rollback in data manipulating SQL statement units can be the occurrence of an exception condition in a data operation. Rollback in transaction units can be the deliberate execution of a ROLLBACK statement by an application program.

Related term: commit

Routine name

The name of a procedure routine.

Related term: Procedure routine

Row

A row is one of the components of a table. In relational databases, data is expressed in terms of two-dimensional tables containing rows and columns.

Row ID

A row ID is used to identify a unique row of a table in the database. Users can manipulate the data of a desired row by using the row ID which is fetched by executing a single row SELECT statement or FETCH statement.

Schema

A constituent element of a database. SymfoWARE/RDB creates tables and view tables by analyzing data using information system analysis technique AA/BRMODELLING.

Schema definition

Schema definition is logical structure definition which defines the base tables and view tables that constitute a database, the column configuration of each table, and the attributes of each column. Authorities, procedure routines, and triggers can be defined as well as tables.

Scope

When using SQL statements for data manipulation to access a table, the table can be accessed within a specified data manipulation range. The specified data manipulation range is called a scope. The scope function can determine a data manipulation range for each user by applying/releasing a scope to/from users who access tables.

Search condition

The specification for identifying rows that are the subject of the operation in data manipulation SQL. For example, a search condition is specified in the WHERE clause of an SQL SELECT statement.

Select column list

SQL statement query specifications and single row SELECT statements specify columns to be targeted for data reference from the table specified in the table expression. The columns to be targeted are specified in a select column list. Column names, set functions, and operation expressions can be specified in a select column list.

Select target list

A select target list is specified in a single row SELECT statement to fetch values to an application program from the columns specified in the select column list. The select target list follows INTO, and specifies host variables in a sequence corresponding to the sequence of columns.

Sequence

Function used to generate values that are unique in the system. Specifying a sequence in the SQL statement enables the generated value to be used. Its main use is to create the primary key values of a table.

Session

The period from the time an application program first accesses SymfoWARE/RDB to the time it terminates. The period from the time a connection is made until disconnection occurs constitutes one session.

Set function

In SQL, functions known as set functions can be used in data manipulation. Five set functions are available: COUNT, AVG, MAX, MIN, and SUM. The COUNT function calculates the row count for tables or columns. The AVG function calculates the average value for a column. The MAX function calculates the maximum value for a column. The MIN function calculates the minimum value for a column. The SUM function calculates the total for a column.

Set operation

In SQL, operations can be performed between groups of rows contained in two tables using a set operation. Three types of set operation are available: union, except, and intersect. Union refers to fetching all rows of both tables. Except refers to fetching rows contained in one table but not in the other. Intersect refers to fetching rows that contain the same value in both tables. Generally, if the result of a set operation is duplicate rows, the data is fetched as one row. However, if necessary, duplicate rows can be fetched as the result without any modification.

Single row SELECT statement

The single row SELECT statement is an SQL data manipulation statement used to reference data. The single row SELECT statement can specify a search condition and fetch one row of data from a table.

Sort specification

When an SQL statement for manipulating data gets the results of a query expression using a cursor, it can rearrange the order of the rows. A sort specification specifies the column that is the sort key and ascending (ASC) or descending (DESC) order. A sort specification is specified in the ORDER BY clause of a cursor declaration.

SQL

SQL is a standard database language for performing database definition and data manipulation. RDB II basically conforms to the protocols of the international standards IS 9075 and ANSI X3.135.

SQL embedded host program

In an application program that manipulates a database, the parts of the program that perform database processing are written using SQL statements. The parts of the program that perform other kinds of processing are written using a programming language such as C language or COBOL. Application program in which SQL statements are embedded are called SQL embedded host programs. The following terms are used for application programs that contain embedded SQL statements:

a. SQL embedded C program
b. SQL embedded COBOL program

SQL embedded host program is the generic term for a) and b).

SQL variable

A variable declared using a procedure routine.

Related term: Procedure routine

SQL variable name

The name of an SQL variable.

Related term: SQL variable

SQLDA structure

A user-defined descriptor area used to store dynamic parameter specification information and select list information contained in a prepared statement. The user sets the values of the dynamic parameter specifications and fetches the results of select lists via this area.

Status variable

The status variable, SQLSTATE, reports the processing result of an SQL statement to an application program. When an SQL statement is executed, the status code for the execution result is stored in the status variable.

Storage structure

A database structure along with logical structure and physical structure. Storage structure physically locates data logically expressed as rows and columns of tables in a database as storage data. A storage structure is expressed using DSO and DSI. The storage structures for base tables are RANDOM structures and SEQUENTIAL structures; BTREE structures are the storage structures for indexes.

Related terms: physical structure and logical structure

Storage structure definition

The definition of mappings between tables and database spaces. The two kinds of storage structure definition are data structure organization (DSO) definition and data structure instance (DSI) definition.

Subquery

A subquery is a specification used in an IN, comparison, quantified, or EXISTS predicate. The specification format is the same as that for a query specification. However, only one column must be specified for the derived table.

Table

In a relational database, data is represented using two-dimensional tables consisting of rows and columns. The two kinds of tables are base tables and view tables.

Table declaration

A table declaration declares the schema in which a table is located. If a table declaration is specified, the schema name need not be specified for a table in data manipulation statements. Using a table declaration simplifies the specification of table names. Using a table declaration also helps to make an application program independent of a data base.

Table expression

In single row SELECT statements and query specifications, the part that specifies the table, row, and group for which data is to be manipulated is the table expression. A table expression may contain the FROM, WHERE, GROUP BY, and HAVING clauses.

Table name

A name attached to a table. Table names are set in schema definitions. Table names are used to specify the tables to be made the subjects of operations in SQL statements that manipulate data.

Target specification

A target specification is used to fetch values stored in a database to an application program. The target specification is specified as a variable.

Trigger definition

A trigger definition defines the data manipulation (insertion) of a table linked to the data manipulation (insertion, deletion, update) of another table.

Unique constraint

A constraint on a table or column. This constraint requires that a table cannot have multiple rows having the same value in a column or combination of columns.

Updatable cursor

In an SQL data manipulation statement, a cursor that can be used to update and delete is called an updatable cursor. The cursor declaration specifies whether a cursor is updatable. For example, if a table specified in the FROM clause of a query specification satisfies just one condition, the cursor is an updatable cursor.

Related term: read-only cursor

UPDATE statement

The UPDATE statement is an SQL data manipulation statement used to update data in rows of tables.

Value expression

A value expression is a column specification, variable specification, set function, or literal, or a specification in which arithmetic operators (+, -, *, and /) are used to combine these items.

Value specification

A value specification, specified from an application program, specifies a value to be stored in, or compared with, a value stored in a data base. A value specification can be specified with a variable specification, literal (numeric, character-string, or national-language character-string), the keyword USER, or a dynamic parameter specification.

WHERE clause

In an SQL data manipulation statement, the WHERE clause is a search condition that specifies rows to be manipulated.